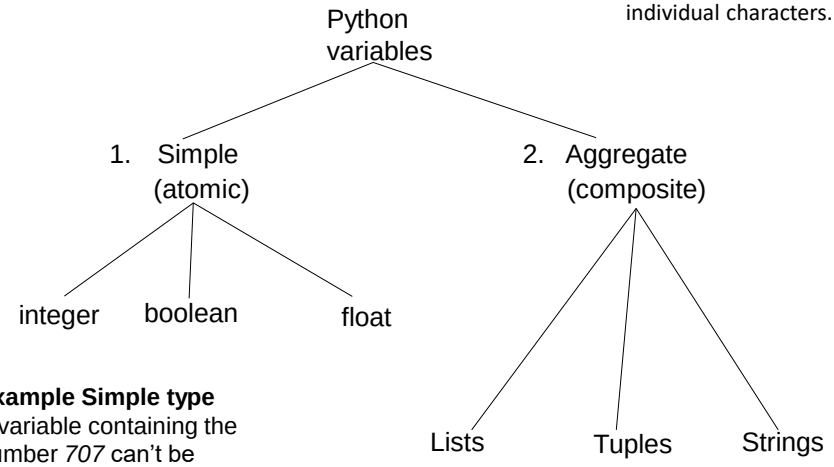
Composite Types, Lists Part 1

- Style: avoiding list bound exceptions (overflow)
- Declaring a list variable
- Accessing a list vs the elements in the list
- Passing lists as parameters
- A new method of parameter passing: pass by reference

James Tam

Types Of Variables

Example composite
A string (sequence of characters) can be decomposed into individual characters.



Example Simple type
A variable containing the number *707* can't be meaningfully decomposed into parts

List

- In many programming languages a list is implemented as an array.
  - This will likely be the term to look for if you are looking for a list-equivalent when learning a new language (i.e. beyond python).
- Python lists have many of the characteristics of the arrays in other programming languages but they also have other features.

Example Problem

- Write a program that will track the percentage grades for a class of students. The program should allow the user to enter the grade for each student. Then it will display the grades for the whole class along with the average.

Why Bother With A List?

- **Name of the example program**: `0classListV1.py`
  - Learning: a "how not to" approach for a solution that should employ lists.

```
CLASS_SIZE = 5

stu1 = float(input("Enter grade for student no. 1: "))
stu2 = float(input("Enter grade for student no. 2: "))
stu3 = float(input("Enter grade for student no. 3: "))
stu4 = float(input("Enter grade for student no. 4: "))
stu5 = float(input("Enter grade for student no. 5: "))
```

Why Bother With A List? (2)

```
total = stu1 + stu2 + stu3 + stu4 + stu5
average = total / CLASS_SIZE

print()
print("GRADES")
print("The average grade is %0.2f%%", %(average))
print("Student no. 1: %0.2f", %(stu1))
print("Student no. 2: %0.2f", %(stu2))
print("Student no. 3: %0.2f", %(stu3))
print("Student no. 4: %0.2f", %(stu4))
print("Student no. 5: %0.2f", %(stu5))
```

Why Bother With A List? (3)

```
total = stu1 + stu2 + stu3 + stu4 + stu5
average = total / CLASS_SIZE

print()
print("GRADES")
print("The average grade is %0.2f%%" %(average))
print("Student no. 1: %0.2f", %(stu1))
print("Student no. 2: %0.2f", %(stu2))
print("Student no. 3: %0.2f", %(stu3))
print("Student no. 4: %0.2f", %(stu4))
print("Student no. 5: %0.2f", %(stu5))
```

NO!

What Were The Problems With The Previous Approach?

- Redundant statements.
- Yet a loop could not be easily employed given the types of variables that you have seen so far.

What's Needed: A List

- A composite variable that is a collection of another type.
  - The composite variable can be manipulated and passed throughout the program as a single entity:
    - Use the name of the "list variable"
    - Example:

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

  - At the same time each element can be accessed individually:
    - Use the name of the list variable and an index.
    - Example:

```
Print(aList[i])
```

Creating A List (Fixed Size)

- **Format ('n' element list):**

Element 0 Element 1 Element n-1

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

Example:

```
#List with 5 elements, index ranges from 0 to (5-1)

percentages = [50.0, 100.0, 78.5, 99.9, 65.1]
```

0 1 2 3 4

Other Examples:

```
letters = ["A", "B", "A"]
names = ["The Borg", "Klingon ", "Hirogin", "Jem'hadar"]
```

1 These 4 names (Borg, Klingon, Hirogin, Jem'hadar) © are CBS

Creating A List (Fixed Size, Same Data In Each Element)

- **Format ('n' element list, n >= 1):**

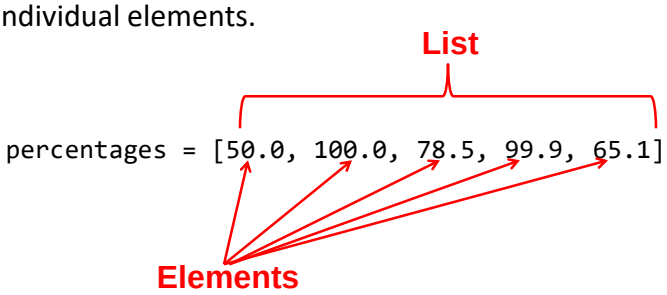
```
<list_name> = [<element data>] * <n>
```

Examples:

```
aList1 = [" "] * 7

# Assume the constant has been declared
aList2 = [-1] * NUMBER_ELEMENTS
```

Accessing A List

- Because a list is composite you can access the entire list or individual elements.



- Name of the list accesses the whole list

```
print(percentages)
```

```
>>> print(percentages)
[50.0, 100.0, 78.5, 99.9, 65.1]
```

- Name of the list and an index "`[index]`"accesses an element

```
print(percentages[1])
```

```
>>> print(percentages[1])
100.0
```

James Tam

Negative Indices

- Although Python allows for negative indices (-1 last element, -2 second last…-<size>) this is unusual and this approach is not allowed in other languages.
- So unless otherwise told your **index should be a positive integer** ranging from <zero> to <list size – 1>
- **Don't use negative indices**.

James Tam

Revised Version Using A List

- **Name of the example program:** `1classListV2.py`
  - Learning: an alternative implementation that illustrates the advantages of using a list. Can access individual elements as well as the entire list.

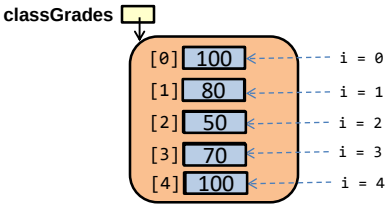
```
CLASS_SIZE = 5

def initialize():
    classGrades = [-1] * CLASS_SIZE
    return(classGrades)
```

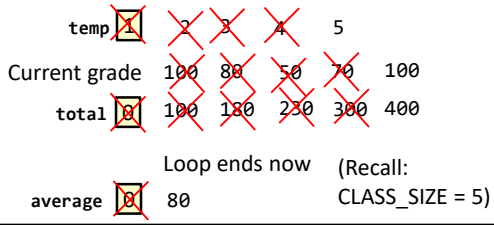
Revised Version Using A List (2)

```
def read(classGrades):
    total = 0
    average = 0
    for i in range (0, CLASS_SIZE, 1):
        temp = i + 1
        print("Enter grade for student no.", temp, ":")
        classGrades[i] = float(input (">"))
        total = total + classGrades[i]
    average = total / CLASS_SIZE
    return(classGrades, average)
```

```
Enter grade for student no. 1 :
>100
Enter grade for student no. 2 :
>80
Enter grade for student no. 3 :
>50
Enter grade for student no. 4 :
>70
Enter grade for student no. 5 :
>100
```

After 'initialize': before loop





Revised Version Using A List (3)

```
def display(classGrades, average):
    print()
    print("GRADES")
    print("The average grade is %0.2f%%" %(average))
    for i in range (0, CLASS_SIZE, 1):
        temp = i + 1
        print("Student No. %d: %0.2f%%"
           %(temp,classGrades[i]))
```

```
GRADES
The average grade is 80.00%
Student No. 1: 100.00%
Student No. 2: 80.00%
Student No. 3: 50.00%
Student No. 4: 70.00%
Student No. 5: 100.00%
```

James Tam

Revised Version Using A List (4)

```
def start():
    classGrades = initialize()
    classGrades, average = read(classGrades)
    display(classGrades,average)

start()
```

James Tam

Creating A List (Variable Size)

- Step 1: Create a variable that refers to the list (list is empty)
- **Format:**

  <list name> = []

- **Example:**

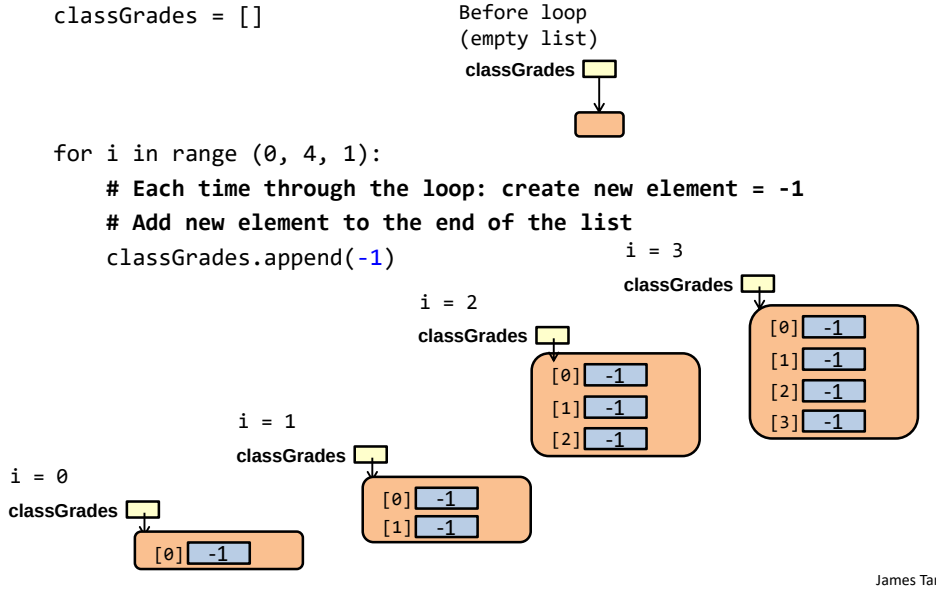
  classGrades = []

Creating A List (Variable Size: 2)

- Step 2: Initialize the list with the elements
- **General format:**
  - Within the body of a loop create each element and then add the new element on the end of the list ('append').
  - The difference between the previous approach (e.g. `aList1 = [" "] * 7`) and this approach is that **new elements need not be all the same**.

Creating A Variable Sized List, Data *Can* Vary: Example

```
classGrades = []

for i in range (0, 4, 1):
    # Each time through the loop: create new element = -1
    # Add new element to the end of the list
    classGrades.append(-1)
```



James Tam

Further Revised Version Using A Dynamically Created List

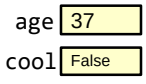
- **Name of the example program:** `2classListV3.py`
  - Learning: creating a list dynamically (one element at a time rather than all at once).

```
CLASS_SIZE = 5

def initialize(): #This is the only function that differs
 classGrades = []
 for i in range (0, CLASS_SIZE, 1):
     classGrades.append(i*10)
 return(classGrades)
```

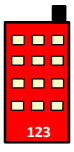
More Details On Lists

- With the simple variable types (integer, float, boolean) you can think of as a single memory location.
  - E.g.

```
age = 37
cool = False
```



- Declaring a list variable will result in two memory locations allocated in memory.
  - One location is for the list itself ("The multi-suite building")



  - Another location "refers to" or contains the address of the building.

James Tam

Example: Illustrating List References

- **Name of the example program:** `3listReferences.py`

```
num = 123
list1 = [1,2,3]
list2 = list1
List1[0] = 888
List2[2] = 777
print(list1)
print(list2)
```

James Tam

One Part Of The Previous Example Was Actually Unneeded

```
def read(classGrades):

        :               :

    return(classGrades, average)
```

When list is passed as a parameter…

…returning the list is likely not needed

More details on 'why' coming up shortly!

Passing A List As A Parameter

- **A reference to the list is passed**, in the function **a local variable** which is another reference can allow **access to the list**.
  - Recall: a reference ~a piece of paper containing an address so this is like having two "pieces of paper" that refer to the same address.
- Example:

```
def read(classGrades):
    ...
    for i in range (0, CLASS_SIZE, 1):
        temp = i + 1
        print("Enter grade for student no.", temp, ":")
        classGrades[i] = float(input (">"))
        total = total + classGrades[i]
def start():
    classGrades = initialize()
    read(classGrades)
```

James Tam

Example: Passing Lists As Parameters

- **Name of the example program**: `4listParametersPassByReference.py`
  - Learning : a list parameter allows changes to the original list (persist even after the function ends).

```
def fun1(aListCopy):
    aListCopy[0] = aListCopy[0] * 2
    aListCopy[1] = aListCopy[1] * 2
    return(aListCopy)

def fun2(aListCopy):
    aListCopy[0] = aListCopy[0] * 2
    aListCopy[1] = aListCopy[1] * 2
```

James Tam

Example: Passing Lists As Parameters (2)

```
def start():
    aList = [2,4]
    print("Original list in start() before function
          calls:\t", end="")
    print(aList)
    aList = fun1(aList)
    print("Original list in start() after calling fun1():\t",
          end="")
    print(aList)
    fun2(aList)
    print("Original list in start() after calling fun2():\t",
          end="")
    print(aList)
start()
```

```
Original list in start() before function calls: [2, 4]
Original list in start() after calling fun1():  [4, 8]
Original list in start() after calling fun2():  [8, 16]
```

James Tam

Passing References (Lists): "Pass-By-Reference"

- Recall: A list variable is actually just a reference to a list (~a paper with an address written on it).

```
aList          = [1,2,3]
```

Reference to the list (contains the memory address)

The list (no name just a location in memory)

- A copy of the address is passed into the function (~copying what's on the paper)

```
def fun(copyList):
    copyList[0] = 10
```

- The local reference 'refers' to the original list (thus the term 'pass-by-reference).
  - Use the paper to go to the specified address.

James Tam

Passing References: Don't Do This

- When passing parameters never (or at least almost never) assign a new value to the reference.
- Example

```
def fun(aReference):
    # Don't do, creates a new list, didn't change the
    # original list
    aReference = [3,2,1]
def start():
    aReference = [1,2,3]
    fun(aReference)
    print(aReference)
```

- Recall: Assignment and using square brackets creates a new list

```
aList = [1,2,3] # Fixed size list, 3 elements
aList = [] # Empty list
```

James Tam

Passing Parameters Which Aren't Lists (Pass By Value)

- A copy of the value stored in the variable is passed into the function.
- Changes made to the parameters are only made to local variables.
- The changed local variables must have their values back to the caller in order to be retained.

James Tam

Example: Passing By Value

- **Name of the example program**:
  `5otherParametersPassByValue.py`
  - Learning: how simple types (integer, float, Boolean) are passed by value (value copied into a local variable)

```
def fun1(aNum,aBool):
    aNum = 21
    aBool = False
    print("In fun1:", aNum,aBool)

def fun2(aNum,aBool):
    aNum = 21
    aBool = False
    print("In fun2:", aNum,aBool)
    return(aNum,aBool)
```

Example: Passing By Value (2)

```
def start():
    aNum = 12
    aBool = True
    print("In start:", aNum,aBool)
    fun1(aNum,aBool)
    print("After fun1:", aNum,aBool)
    aNum,aBool = fun2(aNum,aBool)
    print("After fun2:", aNum,aBool)

start()
```

James Tam

Why Are References Used?

- It looks complex
- Most important reason why it's done: efficiency
  - Since a reference to a list contains the address of the list it allows access to the list.
  - As mentioned if the list is large and a function is called many times the allocation (creation) and de-allocation (destruction/freeing up memory for the list) can reduce program efficiency.
- Type size of references ~range 32 bits (4 bytes) to 64 bits (8 bytes)
- Contrast this with the size of a list
  - E.g., a list that refers to online user accounts (each account is a list element that may be multi-Giga bytes in size).
  - Contrast passing an 8 byte reference to the list vs. passing a multi-Gigabyte list.

James Tam

"Simulation": What If A List And Not A List Reference Passed: Creating A New List Each Function Call

- **Name of example program**: `6listExampleSlow.py`
  - Learning: approximating the speed difference between passing by value vs. passing by reference (simulated pass by value)

```
ONE_HUNDRED_MILLION = 100000000

def fun(i):
    print("Number of times function has been called #%d"
      %(i))
    aList = ["*"] * ONE_HUNDRED_MILLION

def start():
    for i in range (0,ONE_HUNDRED_MILLION,1):
        fun(i)

start()
```

James Tam

Passing Reference And Not Entire List

- **Name of example program**: `7listExampleFast.py`
  - Learning: approximating the speed difference between passing by value vs. passing by reference (actual pass by reference)

```
ONE_HUNDRED_MILLION = 100000000

def fun(aList,num):
    print("fun #%d" %num)

def start():
    aList = ["a"]* ONE_HUNDRED_MILLION
    for i in range(0,ONE_HUNDRED_MILLION,1):
        fun(aList,i)

start()
```

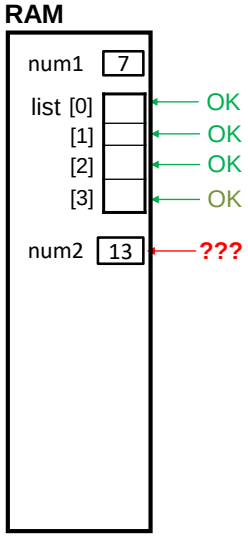
James Tam

Take Care Not To Exceed The Bounds Of The List

Example: `8listBounds.py`

(This example isn't properly implemented to check for list bounds).

```
num1 = 7
list = [0, 1, 2, 3]
num2 = 13
for i in range (0, 4, 1):
    print (list [i])


print()
print(list [4])  ←???
```



A Common Way To Avoid Overflowing A List

- Use a **constant** in conjunction with the list.

```
SIZE = 100
```

- The value in the constant controls traversals of the list

```
for i in range (0, SIZE, 1):
     myList[i] = int(input ("Enter a value:" ))

for i in range (0, SIZE, 1):
    print(myList [i])
```

A Common Way To Avoid Overflowing A List (2)

- Use a constant in conjunction with the list.

```
SIZE = 100000
```

- The value in the constant controls traversals of the list

```
for i in range (0, SIZE, 1):
     myList [i] = int(input ("Enter a value:" ))

for i in range (0, SIZE, 1):
    print (myList [i])
```

A Python Specific Approach To Avoid Overflow

- Use the length function `len` to get the length of list.
  - Since a function call requires some resources/time it's a bit more efficient to store the length in a variable.
  - Unless the length of the list changes refer to the variable rather than calling function again.
- Example:

```
myList = someFunctionCreatesList()
myListLength = len(myList)
i = 0
while (i < myListLength):
    print(myList[i])
```

James Tam

After This Section You Should Now Know

- Techniques to avoid overflowing the bounds of a list
- The difference between a simple vs. a composite type
- Why and when a list should be used
- How to create and initialize a list (each element can be different or is identical)
- How to access or change the elements of a list
- The difference between the parameter passing mechanisms: pass by value vs. pass by reference
  - How are lists passed as parameters