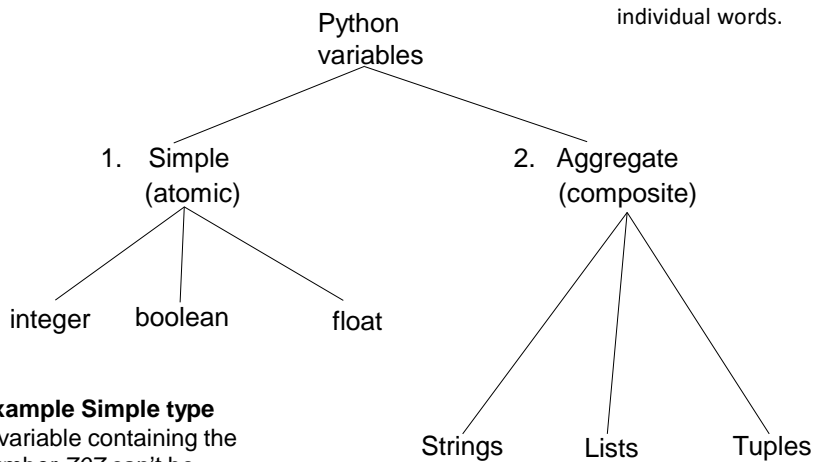# Composite Types

You will learn how to create new variables that are collections of other entities

---

Types Of Variables

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

Python variables

1. Simple (atomic)

2. Aggregate (composite)

integer   boolean   float

Strings   Lists   Tuples

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

Addresses And References

- Real life metaphor: to determine the location that you need to reach the 'address' must be stored (electronic, paper, human memory)
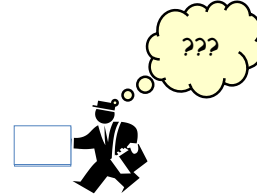
121   122   123   ???

- Think of the delivery address as something that is a 'reference' to the location that you wish to reach.
  – Lose the reference (electronic, paper, memory) and you can't 'access' (go to) the desired location.
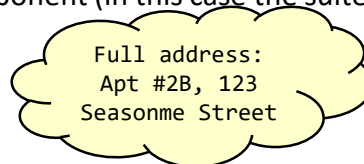
James Tam

Addresses And References (2)

- Composites are analogous to apartment blocks where there is an address of the building but somehow there must be some way to identify each component (in this case the suites).

#3A   #3B

Full address: Apt #2B, 123 Seasonme Street

#2A   #2B

#1A
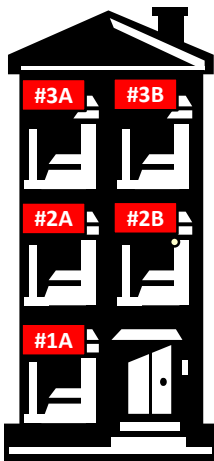
The apartment block is a composite that can be broken down into meaningful parts: the individual suites

123 Seasonme street

James Tam

Small Example Programs Using Strings

- Basic string operations/concepts (some may have already been covered)
  - String1.py (strings as sequences test for inclusion using 'in')
  - String2.py (iterating strings using the 'in' operator)
  - String3.py (concatenation, repetition)
  - String4.py: (passing a whole string to a function)
  - String5.py (indexing the parts of a string)
  - String6.py (demonstrating the immutability of strings)
  - String7.py (converting to/from a string)

Small Example Programs Using Strings (2)

- New/more advanced string examples
  - String8.py (string slicing)
  - String9.py (string splitting)
  - String10.py (determining the size of strings)
  - String11.py (testing if strings meet certain conditions)
  - String12.py (ASCII values of characters)
  - String13.py (Passing strings as parameters – passing a composite)
  - string14.py (using string functions that return modified versions of a string)
  - string15.py (string search functions)

James Tam

Small Example Programs Using Strings (3)

- All the examples will be located in UNIX under:
  /home/231/examples/composites
- Also they can be found by looking at the course website under the URL:
  - http://pages.cpsc.ucalgary.ca/~tamj/231/examples/composites

James Tam

Basic String Operations / Functions

- Some of these may have already been covered earlier during the semester

James Tam

Strings Can Be Conceptualized As Sequences

- The 'in' and 'not in' operations can be performed on a string.

```
User name: username
User name already taken, enter a new one
```

```
User name: xxx
User name already taken, enter a new one
```

- Branching (example name: "string1.py")

```
userNames = "aaa abc username xxx"
userName = input ("User name: ")
if userName in userNames:
    print("User name already taken, enter a new one")
```

```
User name: charlie_sheen
```

- Looping (iterating through the elements: example name "string2.py"[1])

```
sentence = "by ur command"
for temp in sentence:
    print("%s-" %temp, end="")
```

```
b-y- -u-r- -c-o-m-m-a-n-d-
```

String Operations: Concatenation & Repetition

- Concatenation ('+'): connects two or more strings
- Repetition ('*'): repeat a series of characters
- Complete online example: string3.py

```
s1 = "11"
s2 = "17"
s3 = s1 + s2
s4 = s2 * 3
print(s3)    1117
print(s4)    171717
```

James Tam

String: Composite

- Strings are just a series of characters (e.g., alpha, numeric, punctuation etc.)
  - A string can be treated as one entity.
  - Online example: "string4.py"

  ```
  def fun(aString):
      print(aString)
  ```
  `By your command`

  ```
  # START
  aString = "By your command"
  fun(aString)
  ```

- Individual elements (characters) can be accessed via an index.
  - Online example: "string5.py"
  - Note: A string with 'n' elements has an index from 0 to (n-1)

  ```
  aString = "hello"
  print (aString[1])
  print (aString[4])
  ```

  `[csl composites 63 ]> python string5.py`
  `e`
  `o`

Mutable, Constant, Immutable,

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

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

  num `17`

- Constants
  - Memory location *shouldn't* change (Python): may produce a logic error if modified
  - 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.

  ```
  immutable = 12
  immutable = 17
  ```

  immutable `12`

  `17`

  James Tam

Strings Are Immutable

- Even though it may look a string can change they actually cannot be edited (original memory location cannot change).
  - Online example: "string6.py"

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

Converting To Strings

- Online example: string7.py

```
a = 2
b = 2.5
c = a + b   # Addition
print(c)    # Yields 4.5

# str() Converts argument to a String
# Convert to string and then concatenate
c = str(a) + str(b)
print(c)    # Yields '22.5'
```

James Tam

Converting From Strings

```
x = '3'
y = '4.5'
# int(): convert to integer
# float(): convert to floating point
# Convert to numeric and then add
z = int(x) + float(y)
print(z)    # Yields 7.5
```

James Tam

Advanced Operations / Functions

- These operations and functions likely have not yet been covered

James Tam

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

- Slicing a string will return a portion of a string based on the indices provided
- The index can indicate the start and end point of the substring.
- **Format**:

  string_name [*start_index* : *end_index*]

- **Online example**: string8.py

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

```
abcdefghij
```
```
cde
```
```
abcde
```
```
hij
```

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.

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**: string9.py
  ```
  aString = "man who smiles"
  # Default split character is a space
  one, two, three = aString.split()
  print(one)
  print(two)
  print(three)
  ```
  ```
  man
  who
  smiles
  ```
  ```
  aString = "James,Tam"
  first, last = aString.split(',')
  nic = first + " \"The Bullet\" " + last
  print(nic)
  ```
  ```
  James "The Bullet" Tam
  ```

James Tam

Determining Size

- The 'len()' function can count the number of characters in a string.

- Example program: string10.py
  ```
  MAX_FILE_LENGTH = 256
  SUFFIX_LENGTH = 3
  ```
  ```
  Enter new file name (max 256 characters): a.txt
  Text file
  2:5 txt
  ```
  ```
  filename = input("Enter new file name (max 256 characters): ")
  if (len(filename) > MAX_FILE_LENGTH):
      print("File name exceeded the max size of %d characters, you bad"
        %(MAX_FILE_LENGTH))
  else:
   # Find file type, last three characters in string e.g., resume.txt
      endSuffix = len(filename)
      startSuffix = endSuffix - SUFFIX_LENGTH
      suffix = filename[startSuffix:endSuffix]
      if (suffix == "txt"):
          print("Text file")
          print("%d:%d %s" %(startSuffix,endSuffix,suffix))
  ```

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)

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.

Applying A String Testing Function

Name of the online example: "string11.py"

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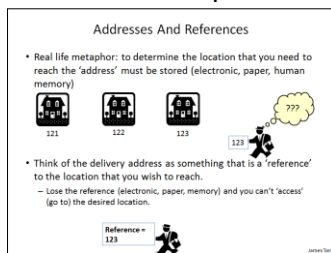
```
Enter an integer: abc
abc is not an integer
```

```
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)

Strings And References

• Recall the concepts of references and addresses

Addresses And References

• Real life metaphor: to determine the location that you need to reach the 'address' must be stored (electronic, paper, human memory)

• Think of the delivery address as something that is a 'reference' to the location that you wish to reach.
  – Lose the reference (electronic, paper, memory) and you can't 'access' (go to) the desired location.
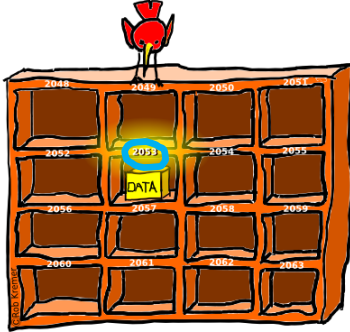
Reference = 123

James Tam

  – A reference contains an address

• A String 'variable' does not directly contain the contents of a string
  – Instead the string contains the address ("refers to") of a string

James Tam

Recap: Variables

- Variables are a 'slot' in memory that contains 'one piece' of information.

  ```
  num = 123
  ```


- Normally a location is accessed via the name of the variable.
  - Note however that each location is also numbered!
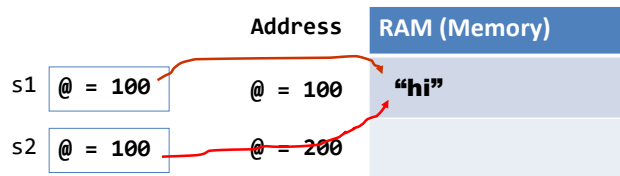
Image: Curtesy of Rob Kremer   James Tam

References And Strings
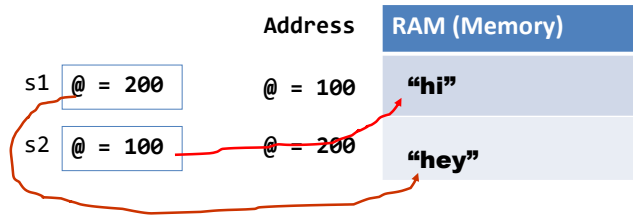
- Example:

  ```
  s1 = "hi"

  s2 = s1
  ```

	Address	RAM (Memory)
s1	@ = 100	"hi"
s2	@ = 100	

James Tam

References And Strings

- Example:

```
s1 = "hi"

s2 = s1

s1 = "hey"
```

| s1 | @ = 200 |
| s2 | @ = 100 |

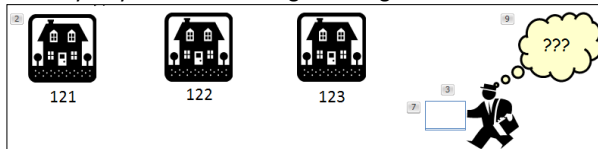| Address	RAM (Memory)
@ = 100	"hi"
@ = 200	"hey"

Note:
- The string and the reference to the string are separate e.g., s1 originally referred to the string "hi" but later it referred to the string "hey"
- The only way to access a string is through the reference

121   122   123   ???

James Tam

References And Strings (NEW)

- The string can ONLY be accessed via an address.
- That is, if there are no references to a string then that string is lost.

User enters: "I am really a secret agent spy"

- Example:

```
secret = input("Tell me your life's secret")
```

| secret | @ = 100 |

Address	RAM (Memory)
@ = 100	"I am really a secret agent spy"
@ = 200	

James Tam

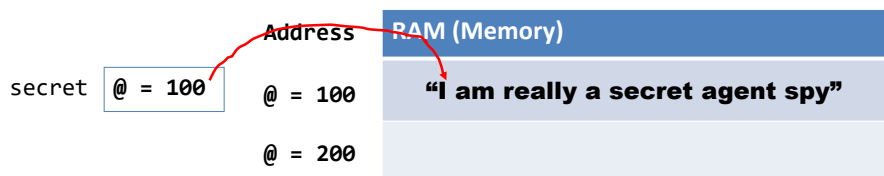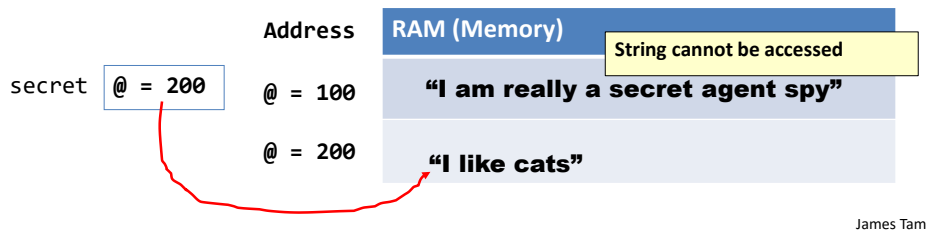Composites   15

References And Strings (NEW)

- The string can ONLY be accessed via an address.
- That is, if there are no references to a string then that string is lost.
- Example:
```
secret = input("Tell me your life's secret")
secret = "I like cats"
print(secret)
```

	Address	RAM (Memory)	
secret	@ = 200		String cannot be accessed
	@ = 100	"I am really a secret agent spy"	
	@ = 200	"I like cats"	

James Tam

ASCII Values

- 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.
- The ord() function can be used to determine the ASCII code for a character (string of length one).
- Example: string12.py
```
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
```

Passing Strings As Parameters

- A string is composite so either the entire string or just a sub-string can be passed as a parameter.
- Full example: `string13.py`

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

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

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

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

Passing whole string

Passing part of a string

James Tam

Functions That Return Modified Copies Of Strings

- 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.

Examples: Functions That Return Modified Copies

Name of the online example: `string14.py`

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

aString = "xxhello there"
print(aString)
aString = aString.lstrip ('x')
print(aString)
aString = "xxhellx thxrx"
aString = aString.lstrip ('x')
print(aString)
```

`talk1! AbouT`

`TALK1! ABOUT`

`xxhello there`

`hello there`

`hellx thxrx`

Functions To Search Strings

Function	Description
`endswith` `  (substring)`	A substring is the parameter and the function returns true only if the string ends with the substring.
`startswith` `  (substring)`	A substring is the parameter and the function returns true only if the string starts with the substring.
`find` `  (substring)`	A substring is the parameter and the function returns the lowest index in the string where the substring is found (or -1 if the substring was not found).

`replace`
`  (oldstring,`
`   newstring)`
	The function returns a copy of the string with all instances of 'oldstring' replace by 'newstring'

Examples Of Functions To Search Strings

Name of the online example: `string15.py`

```
temp = input ("Enter a sentence: ")
if not ((temp.endswith('.')) or
        (temp.endswith('!')) or
        (temp.endswith ('?'))):
    print("Not a sentence")

temp = "XXabcXabcabc"
index = temp.find("abc")
print(index)

temp = temp.replace("abc", "^-^")
print(temp)
```

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.
- 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?

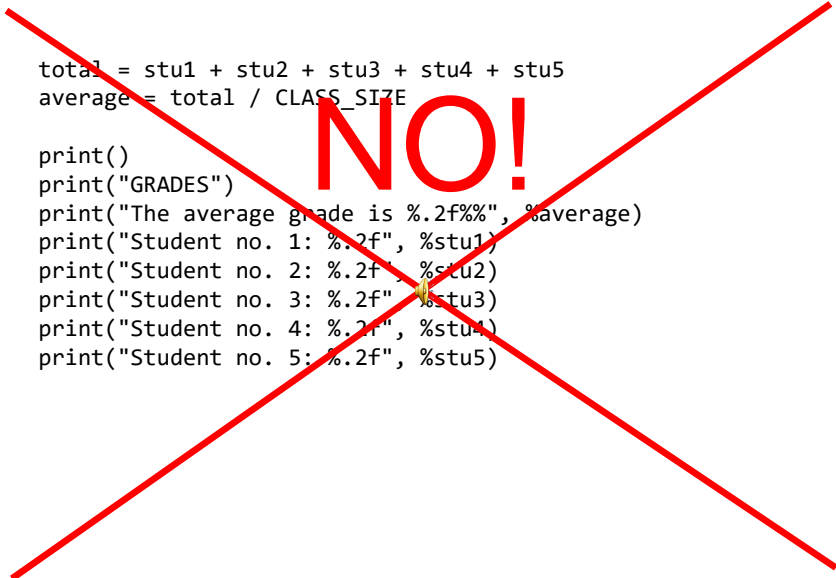- Name of the example program: `classList1.py`

```
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 %.2f%%", %average)
print("Student no. 1: %.2f", %stu1)
print("Student no. 2: %.2f", %stu2)
print("Student no. 3: %.2f", %stu3)
print("Student no. 4: %.2f", %stu4)
print("Student no. 5: %.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 %.2f%%", %average)
print("Student no. 1: %.2f", %stu1)
print("Student no. 2: %.2f", %stu2)
print("Student no. 3: %.2f", %stu3)
print("Student no. 4: %.2f", %stu4)
print("Student no. 5: %.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 composite variable that is a collection of another type.
  - The composite variable can be manipulated and passed throughout the program as a single entity.
  - At the same time each element can be accessed individually.
- What's needed…a list!

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

                       0      1      2      3      4
percentages = [50.0, 100.0, 78.5, 99.9, 65.1]
```

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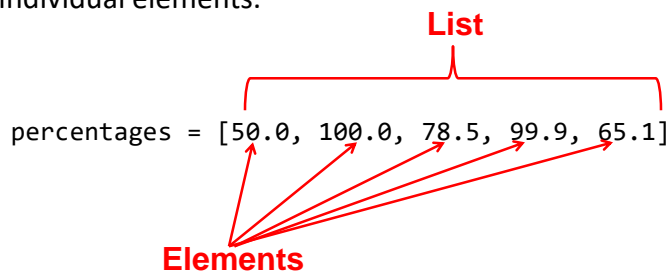
```
letters = ['A', 'B', 'A']
names = ["The Borg", "Klingon ", "Hirogin", "Jem'hadar"]
```

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

Accessing A List

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

List

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

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>

James Tam

Accessing Lists

- Lists can only be accessed through the reference
  ```
  list1 = [1,2,3]
  ```
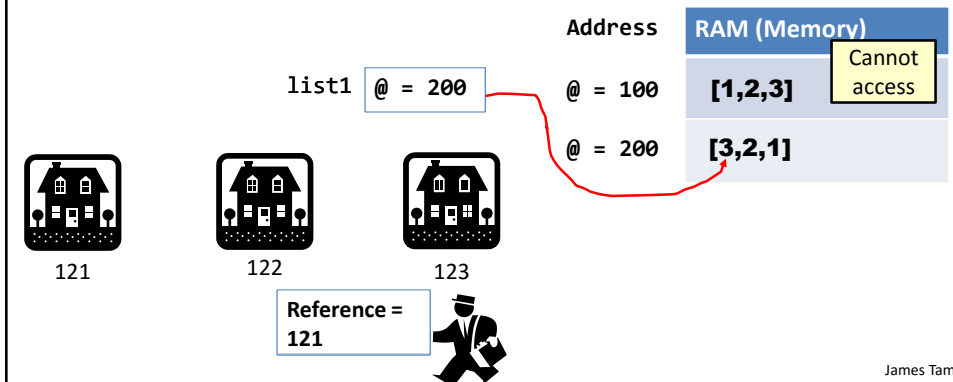
	Address	RAM (Memory)
list1 @ = 100	@ = 100	**[1,2,3]**
	@ = 200	

James Tam

Accessing Lists

- Lists can only be accessed through the reference

```
list1 = [1,2,3]
list1 = [3,2,1]
```

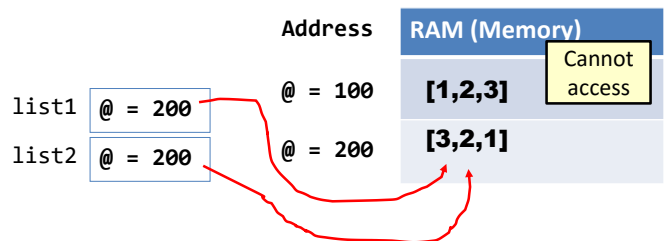|	Address	RAM (Memory)	
list1 @ = 200	@ = 100	[1,2,3]	Cannot access
	@ = 200	[3,2,1]	

121   122   123

Reference = 121

James Tam

Accessing Lists

- Lists can only be accessed through the reference

```
list1 = [1,2,3]
list1 = [3,2,1]
list2 = list1
```

	Address	RAM (Memory)	
	@ = 100	[1,2,3]	Cannot access
list1 @ = 200	@ = 200	[3,2,1]	
list2 @ = 200			

James Tam

Creating A List (Variable Size)

- Step 1: Create a variable that refers to the list
- **Format:**
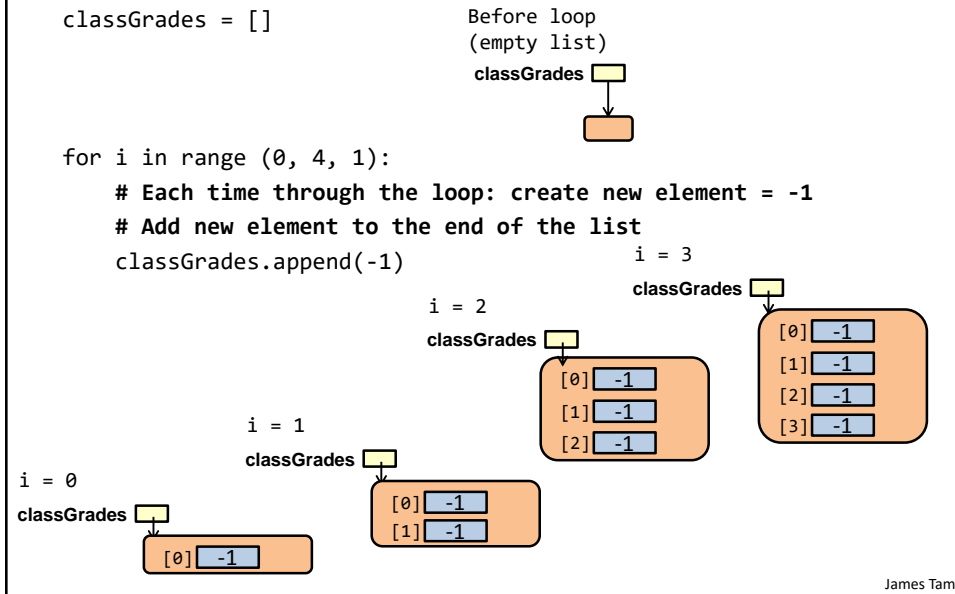   `<list name> = []`

- **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')

Creating A Variable Sized List: Example

```
classGrades = []
```
Before loop
(empty list)

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)
```

i = 3

classGrades ☐

[0]	-1
[1]	-1
[2]	-1
[3]	-1

i = 2

classGrades ☐

[0]	-1
[1]	-1
[2]	-1

i = 1

classGrades ☐

[0]	-1
[1]	-1

i = 0

classGrades ☐

[0]	-1

James Tam

Revised Version Using A List

• **Name of the example program:** classList2.py

```
CLASS_SIZE = 5

def initialize():
    classGrades = []
    for i in range (0, CLASS_SIZE, 1):
        classGrades.append(-1)
    return(classGrades)
```
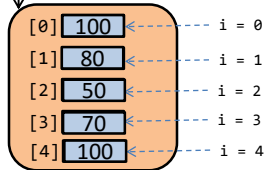
Revised Version Using A List (2)

```
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

classGrades ☐

[0]	100	← --- i = 0
[1]	80	← --- i = 1
[2]	50	← --- i = 2
[3]	70	← --- i = 3
[4]	100	← --- i = 4

temp 1̶ 2̶ 3̶ 4̶ 5

Current grade 100 80̶ 50̶ 70̶ 100

total 0̶ 100̶ 180̶ 230̶ 300̶ 400

Loop ends now   (Recall:

average 0̶ 80   CLASS_SIZE = 5)

Revised Version Using A List (3)

```
def display(classGrades, average):
    print()
    print("GRADES")
    print("The average grade is %.2f%%" %average)
    for i in range (0, CLASS_SIZE, 1):
        temp = i + 1
        print("Student No. %d: %.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

Composites

28

Revised Version Using A List (4)

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


start()
```

One Part Of The Previous Example Was Actually Unneeded

```
def read(classGrades):


         :                    :


    return (classGrades, average)
```

When list is passed as a parameter…
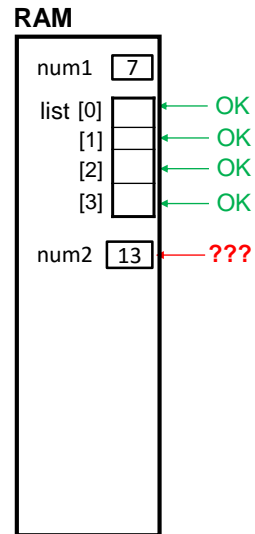
…returning the list is likely not needed

More details on 'why' coming up shortly!

Take Care Not To Exceed The Bounds Of The List

RAM

Example: `listBounds.py`

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


print ()
print (list [4])
```

num1	7	
list [0]		← OK
[1]		← OK
[2]		← OK
[3]		← OK
num2	13	←— ???

print (list [4]) ←— **???**

One Way Of Avoiding An Overflow Of The 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])
```

Composites

One Way Of Avoiding An Overflow Of The List

- 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])
  ```

Lists: Searching/Modifying, Len() Function

- Common problem: searching for matches that meet a certain criteria.
- The matches may simply be viewed or they may modified with a new value.
- Example: listFindModify.py
  ```
  grades = ['A','B','C','A','B','C','A']
  last = len(grades) - 1
  i = 0
  while (i <= last):
      if (grades[i] == 'A'):    # Search for matches
          grades[i] = 'A+'      # Modify element
      i = i + 1
  print(grades)
  ```
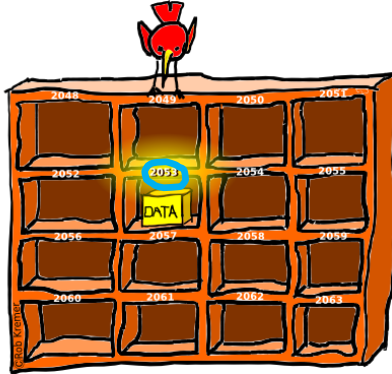
  ```
  ['A+', 'B', 'C', 'A+', 'B', 'C', 'A+']
  ```

James Tam

Recap: Variables

- Variables are a 'slot' in memory that contains 'one piece' of information.

```
num = 123
```


- Normally a location is accessed via the name of the variable.
  - Note however that each location is also numbered!

Recap: Assignment (Simple Types)

```
num1 = 2
num2 = 3
num1 = num2
```

Copy contents from memory location called 'num2' into location called 'num1'

List Variables Are References To Lists (Not Actual Lists)

- Most of the time the difference between a reference to a list and the actual list is not noticeable.
- However there will be times that it's important to make that distinction e.g., using the assignment operator, passing parameters.
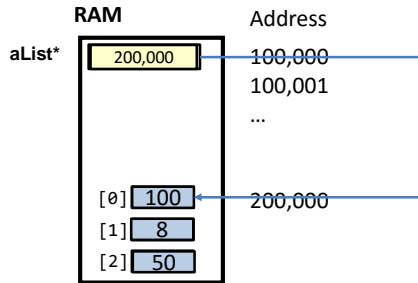- Small example:

```
aList = []
aList = [100,8,50]
```

Create list

Put address in reference

RAM   Address

aList* | 200,000 | 100,000
                        100,001
                        …

[0] | 100 | 200,000
[1] | 8 |
[2] | 50 |

Note
•A reference to a list actually contains an address.
•An 'empty list' contains no address yet
•A non-empty list contains the address of the list

James Tam

Example: List References

```
list1 = [1,2,3]
list2 = list1
# Looks like two lists, actually just two references to one list

print(list1,list2)      [1, 2, 3] [1, 2, 3]
list1 = [3,2,1]
# List1 refers to a new list
print(list1,list2)      [3, 2, 1] [1, 2, 3]
```

James Tam

Copying Lists

- If you use the assignment operator to copy from one list to another you will end up with only one list).
- Name of the example program: `copyList1.py`

```
list1 = [1,2]
list2 = [2,1]
print (list1, list2)
```
```
[1, 2] [2, 1]
```

```
# Two ref to one list
list1 = list2
print (list1, list2)
```
```
[2, 1] [2, 1]
```

```
list1[0] = 99
print (list1, list2)
```
```
[99, 1] [99, 1]
```

Copying Lists (2)

- To copy the elements of one list to another a loop is needed to copy each element.
- Name of the example program: `copyList2.py`

```
list1 = [1,2,3]
list2 = []

for i in range (0, 3, 1):
    list2.append(list1[i])

print(list1, list2)
list1[1] = 99
print(list1, list2)
```
```
[1, 2, 3] [1, 2, 3]
```
```
[1, 99, 3] [1, 2, 3]
```

Composites

Recap: Parameter Passing (Simple Types): Behavior Of "Pass-By-Value"

```
def fun(num):
    print(num)
    num = num + num
    print(num)


def start():
    num = 1
    print(num)
    fun(num)
    print(num)

start()
```

Passing Lists As Parameters

- When a list variable is passed into function it's not actually the whole list that is passed.

```
def start():
    aList = [1,2,3]
    fun(aList)
```

Address of list passed (memory efficient if list is large and the function is called many times)

- Instead it's just a reference to a list (address) that is passed into the function (which then stores the address in a local variable)

```
def fun(aList):
```

The address is stored in a local variable
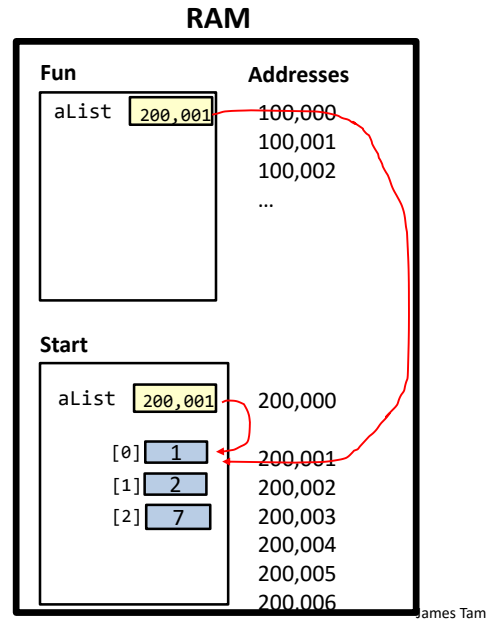
121   122   123   **123**

123

Composites

Passing Lists As Parameters

RAM

```
def fun(aList):
    aList[2] = 7
```

Fun   **Addresses**

aList   200,001 ── 100,000
                    100,001
                    100,002
                    ...

Start

```
def start():
    aList = []

    aList = [1,2,3]

    fun(aList)
```

aList   200,001   200,000

[0] 1   200,001
[1] 2   200,002
[2] 7   200,003
          200,004
          200,005
          200,006

James Tam

Example: Passing Lists As Parameters

• Name of complete example: listParameters.py

```
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"

- A copy of the address is passed into the function.
- The local reference 'refers' to the original list (thus the term 'pass-by-reference).

James Tam

Passing References: Don't Do This

- A reference to the list contains the address of a list
- The address stored in the parameter passed in (calling function) and the local variable that stores the address passed in (function called) both point to the same list.
- Never (or at least almost never) assign a new value to the reference (Advanced questions: What happened? Why?)
- Example

```
def fun(aReference):
    aReference = [3,2,1]  # Don't do this!

def start():
    aReference = [1,2,3]
    fun(aReference)
```

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.

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

- Name of full online example: listExampleSlow.py

```python
MAX = 1000000

def fun(i):
    print("Number of times function has been called %d" %(i))
    aList = []
    for j in range (0,MAX,1):
        aList.append(str(j))


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

start()
```
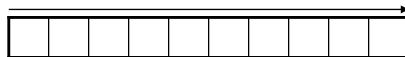

When To Use Lists Of Different Dimensions

- 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

  One dimension (which student)

- (2D list)
  - Expanded grades program
  - 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)

Important:

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

List elements are specified in the order of [row] [column]

Specifying only a single value specifies the row

Columns

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

Rows

Composites

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

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

Name of the example program: `display2DList.py`

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

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

for r in range (0,4, 1):
    for c in range (0,3,1):
        print(matrix[r][c], end="")
    print()

print(matrix[2][0])    #2 not 0
```
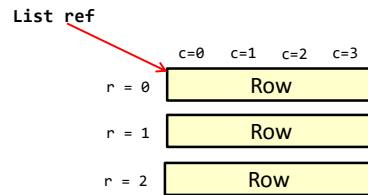
r = 0 `[0, 0, 0]`
r = 1 `[1, 1, 1]`
r = 2 `[2, 2, 2]`
r = 3 `[3, 3, 3]`

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

2

Creating And Initializing A Multi-Dimensional List In Python (3)

General structure (Using loops):

- Create a variable that refers to a 1D list.
- One loop (outer loop) traverses the rows.
- Each iteration of the outer loop creates a new 1D list.
- Then the inner loop traverses the columns of the newly created 1D list creating and initializing each element in a fashion similar to how a single 1D list was created and initialized.
- Repeat the process for each row in the list

```
List ref
                    c=0   c=1   c=2   c=3
         r = 0    │        Row        │
         r = 1    │        Row        │
         r = 2    │        Row        │

                    Etc.
```

Creating And Initializing A Multi-Dimensional List In Python (4)

- **Example (Using loops):**

```
aGrid = []                    # Create a reference to the list
for r in range (0, 3, 1):     # Outer loop runs once for each row
    aGrid.append ([])         # Create an empty row (a 1D list)
    for c in range (0, 3, 1): # Inner loop runs once for each column
        aGrid[r].append (" ") # Create and initialize each element
                              # (space) of the 1D list
```

James Tam

Composites   42

Example 2D List Program: A Character-Based Grid

•**Name of the example program:** simple_grid.py

```
aGrid = []

for r in range (0,2,1):
    aGrid.append ([])
    for c in range (0,3,1):
        aGrid[r].append (str(r+c))

for r in range (0,2,1):
   for c in range (0,3,1):
       print(matrix[r][c], end="")
   print()
```

Quick Note" 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 = ["James", "Tam", "210-9455", 707]

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.
- Tuples are used to store data that should not change.

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: `tuples1.py`

```
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

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 parameters can be passed by reference.

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)

List operations:
  – For a numerical list: implement some common mathematical functions (e.g., average, min, max, mode).
  – 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).

After This Section You Should Now Know

- The difference between a simple vs. a composite type
- What is the difference between a mutable and an immutable type
- How strings are actually a composite type
- Common string functions and operations
- Why and when a list should be used
- How to create and initialize a list (fixed and dynamic size)
- How to access or change the elements of a list
- How to search a list for matches
- Copying lists: How does it work/How to do it properly

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