

# CS65: Introduction to Computer Science

Midterm Review  
Quiz 3



Md Alimoor Reza  
Assistant Professor of Computer Science

# Midterm Exam

- Grading policy:

- *Programming Assignments (25%)*. Homework programming activities.
- *Labs (20%)*. Completing programming activities during class.
- *Quizzes (10%)*. true/false, fill in the blanks, etc.
- *Midterm (15%)*. Paper based exam midway through the semester.
- *Final (20%)*. Paper based exam by the end of the semester.
- *Final project (10%)*. Your proposed group project (2-3 members).

- Grading scale:

- |                 |                  |                  |
|-----------------|------------------|------------------|
| • A (93%-100%)  | • A- (90%-92.9%) | • B+ (87%-89.9%) |
| • B (84%-86.9%) | • B- (80%-83.9%) | • C+ (77%-79.9%) |
| • C (74%-76.9%) | • C- (70%-73.9%) | • D (60%-69.9%)  |
| • F (0%-59.9%)  |                  |                  |

# Topics

- Variables, expression
- Functions
- Scope for local and global variables

- Boolean type and boolean expression
- Selection statements are useful for branching inside your program
  - if
  - if-else
  - if-elif-...-else

- Sequence
  - String
  - List
- The **while** loop
- The **for** loop to solve a repetitive task
  - Value **for** loop
  - Index **for** loop
  - Nested **for** loop

# Variables

- Variable is a named storage space in computer memory for one Python value
  - Either we can write a value into a variable
  - Or we can read the value stored in that variable

```
33 time_sec = 60
34 temp_degree = 27
35
36 mile_to_kilometer = 1.609
37 price_in_dollars = 1500.89
```

- time\_sec, temp\_degree, miles\_to\_kilometer are variables



# Variable and assignment operator

- Need to use assignment operator (=) to store a value
- Location of assignment on the left
- Single value or some calculated value on the right
- **variable\_name = value**

```
33 time_sec = 60
34 temp_degree = 27
35
36 mile_to_kilometer = 1.609
37 price_in_dollars = 1500.89
```

Numbers

```
first_name = "Md Alimoor"
last_name = "Reza"
```

Textual data

# Rules for Variable Naming

- Give meaningful variable name to make it easily readable

```
x = 1.609
```

Vs

```
mile_to_kilometer = 1.609
```

- Name should begin with a lowercase letter
- Use underscore to connect multiple words

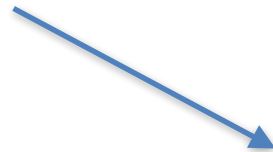
```
milestokilometer = 1.609  
MilesToKilometer = 1.609  
milesToKilometer = 1.609
```

Vs

```
mile_to_kilometer = 1.609
```

# Rules for Variable Naming

- Names can only contain letter, numbers, and underscores
- First character must be a letter or an underscore
  - Then use letter/numbers/underscore
- Cannot be a Python keyword




and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

- Cannot contain spaces
- Variable names are case sensitive
  - Uppercase and lowercase name will signify different variable

# Expression

- A fragment of Python code that calculates a new value called an expression
- For example, you can convert miles into meters using the following expression:

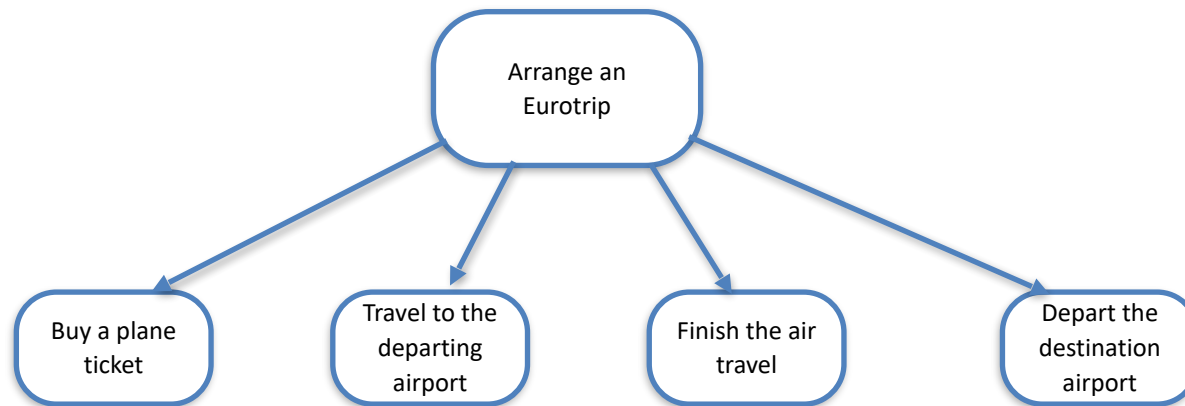


```
num_of_miles = 10  
miles_to_kilometer = 1.609
```

```
num_of_meter = num_of_miles*miles_to_kilometer*1000
```

# Functions

- **Function** is a sequence of statements that performs a specific task — also called a subroutine
- Decompose a bigger task with the help of several smaller subtasks



**You can write a python function for individual subtask!**

# Why should you use Functions?

- Decompose a bigger task with the help of several smaller subtasks
  - Code becomes more modular and manageable
    - Imagine, you have to write the same calculations over and over again eg 100 times!
  - Code for a subtask can be reusable
  - Individual member in a team can write different functions
  - Improves code readability

# Functions

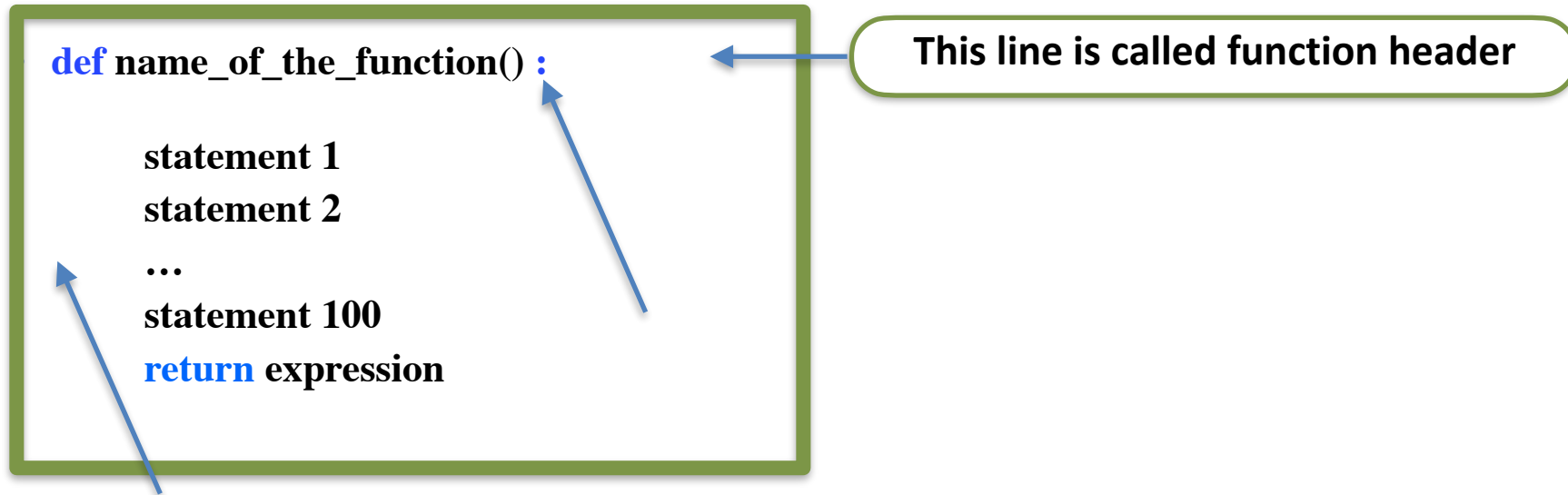
- **Function** is a sequence of statements that performs a specific task
  - **Define** a function once
    - formula or template to solve a task with a series of statements
    - definition **doesn't do** anything unless it is called
  - **Call** function as many times as you like & receive return values
    - supply a matching signature to invoke an already defined function
- Two types of functions:
  - **User-defined** function
    - you define then call it
  - **Built-in** function
    - it is already out there, just call.

# Super Important (User-defined function): function defining vs function calling

- User-defined functions
  - **defining** function: what statements it will execute
  - **calling** function: invoke/execute the defined body



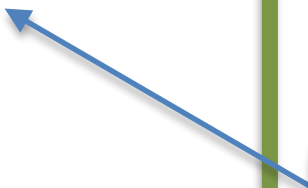
# Define a Function with no Parameters



- **name\_of\_the\_function**: a meaningful name denoting the task with a preceding **def** keyword
- **statements**: a sequence of python instructions to be executed followed by an optional **return** keyword with expression(s)
  - without a **return** statement function implicitly returns **None**
- Notice: indention (eg, tab) is required to define a **function** and also notice at the end of the condition expression there is a **colon**

# Define a Function with Parameters

```
def name_of_the_function(param1, param2, ..., param4) :  
  
    statement 1  
    statement 2  
    ...  
    statement 100  
    return expression
```



Parameters

- Add a number of **parameters** as required for your task:
  - Parameters are variables used to exchange values during function call
  - Values are mapped to parameters each time the function is called
  - Parameters are not available outside the function

# User defined function example

```
1 # Author's name: Md Alimoor Reza
2 # Author's contact: md.reza@drake.edu
3 # Date: (September 7, 2021)
4 # Collaborator:
5 #     self
6
7
8 # this user defined function adds two numbers
9 def add_numbers(num1, num2):
10     sum = num1 + num2
11     return sum
12
13
```

Shell x

```
>>> a = 1
>>> b = 2
>>> res = add_numbers(a, b)
>>> print("sum of", a, "and", b, ":", res)

sum of 1 and 2 : 3
```

Watch out for these items!

# Calling a Function

- **name\_of\_the\_function**(*argument<sub>1</sub>*, *argument<sub>2</sub>*, ..., *argument<sub>4</sub>*)

## Defining a function

```
# this user defined function adds  
def add_numbers(num1, num2):  
    sum = num1 + num2  
    return sum
```

## Parameters

## Calling a function

```
Shell ×  
>>> res1 = add_numbers(1, 3)  
>>> res1 = add_numbers(100, 5)  
>>> res1 = add_numbers(50000, 123)  
>>>
```

## Arguments

- Function **calling name** should match function **definition name**
- Use *values*, *expression*, or *variables* to the **parameters** of the function
  - **arguments** should match **parameters**: one-to-one mapping
- When you call the function the execution gets transferred to the statements inside the function definition

# Calling with values

```
1 # Author's name: Md Alimoor Reza
2 # Author's contact: md.reza@drake.edu
3 # Date: (September 7, 2021)
4 # Collaborator:
5 #     self
6
7
8 # this user defined function adds two numbers
9 def add_numbers(num1, num2):
10     sum = num1 + num2
11     print("add function: called with num1=%d num2=%d and res=%d"%(num1,num2, num1+num2))
12     return sum
13
14 def sub_numbers(num1, num2):
15     sub = num1 - num2
16     print("subtract function: called with num1=%d num2=%d and res=%d"%(num1,num2, num1-num2))
17     return sub
18
19 def mul_numbers(a,b):
20     # Your task
21     return
22 def div_numbers(a,b):
23     # Your task
24     return
25
```

**Defining a function**

Shell x

```
Python 3.7.9 (bundled)
>>> %Run lec3_demo2.py
>>> sub = sub_numbers(10, 4)

subtract function: called with num1=10 num2=4 and res=6

>>> print("result of subtraction from %d to %d is %d"%(10,4, sub))

result of subtraction from 10 to 4 is 6
```

**Calling a function**

# Calling with variables

```
1 # Author's name: Md Alimoor Reza
2 # Author's contact: md.reza@drake.edu
3 # Date: (September 7, 2021)
4 # Collaborator:
5 #     self
6
7
8 # this user defined function adds two numbers
9 def add_numbers(num1, num2):
10     sum = num1 + num2
11     print("add function: called with num1=%d num2=%d and res=%d"%(num1,num2, num1+num2))
12     return sum
13
14 def sub_numbers(num1, num2):
15     sub = num1 - num2
16     print("subtract function: called with num1=%d num2=%d and res=%d"%(num1,num2, num1-num2))
17     return sub
18
19 def mul_numbers(a,b):
20     # Your task
21     return
22 def div_numbers(a,b):
23     # Your task
24     return
25
```

Calling the same function  
with variables

Shell x

```
>>> a = 10
>>> b = 4
>>> sub = sub_numbers(a, b)

subtract function: called with num1=10 num2=4 and res=6

>>> print("result of subtraction from %d to %d is %d"%(a,b, sub))

result of subtraction from 10 to 4 is 6
```

# Calling a function multiple times

```
1 # Author's name: Md Alimoor Reza
2 # Author's contact: md.reza@drake.edu
3 # Date: (September 7, 2021)
4 # Collaborator:
5 #     self
6
7
8 # this user defined function adds two numbers
9 def add_numbers(num1, num2):
10     sum = num1 + num2
11     print("add function: called with num1=%d num2=%d and res=%d"%(num1,num2, num1+num2))
12     return sum
13
14 def sub_numbers(num1, num2):
15     sub = num1 - num2
16     print("subtract function: called with num1=%d num2=%d and res=%d"%(num1,num2, num1-num2))
17     return sub
18
```

Calling the function multiple times

Shell x

result of subtraction from 10 to 4 is 6

>>>

Python 3.7.9 (bundled)

>>> %Run lec3\_demo2.py

>>> sub1 = sub\_numbers(10, 4)

subtract function: called with num1=10 num2=4 and res=6

>>> sub2 = sub\_numbers(10, 5)

subtract function: called with num1=10 num2=5 and res=5

>>> sub3 = sub\_numbers(10, 6)

subtract function: called with num1=10 num2=6 and res=4

>>>

# Built-in function

- Built-in function in Python *input*("....")
  - Step 1: displays the prompt to the user
  - Step 2: waits for user to type in something
  - Step 3: returns the typed content when user hits enter
  - Step 4: this value is stored if assigned to a variable

```
rect_a = input("enter the length of rectangle side a: ")  
print(rect_a)
```



# Built-in function examples

- You **do not** need to **define** the function; just call it
- We have already used 3 built-in functions:

- *print()*

```
>>> print("hello world.")  
hello world.
```

- *int()*

```
>>> b = 12.56  
>>> c = int(b)  
>>> print("converted integer number is ", c)  
converted integer number is 12
```

# Other built-in functions

- If you want to use not so commonly available built-in functions, those built-in functions need to be imported using `import` keyword from a library
  - library also called a module
- Import the **module** before using it usually at the top of your python file
- Call function using *module\_name . function\_name*

```
import math  
value_of_pi = math.pi
```

# Module

- Formally, a module is a component containing Python functions, variables or class
- Each python file (with \*.py) is a module
- They need to be imported from a module using `import`
  - Several ways of importing module components

<https://docs.python.org/3/tutorial/modules.html>

# Module import variations

Explicitly need to use ***math.pi*** or ***math.sin***

```
# ----- Module import variation 1 -----  
import math  
  
# variables initialization  
angle_in_degree = 45  
angle_in_rad = value_of_pi*angle_in_degree/180.0  
  
# calculation  
value_of_pi = math.pi  
var2 = math.sin(angle_in_rad)  
  
print("sin(", angle_in_degree,") is ", var2)
```

Directly access ***pi*** and ***sin*** but nothing else

```
# ----- Module import variation 3 -----  
from math import pi  
from math import sin  
  
# variables initialization  
angle_in_degree = 45  
value_of_pi      = pi  
angle_in_rad     = value_of_pi*angle_in_degree/180.0  
var2             = sin(angle_in_rad)  
  
print("sin(", angle_in_degree,") is ", var2)
```

```
# ----- Module import variation 2 -----  
from math import *  
  
# variables initialization  
angle_in_degree = 45  
value_of_pi     = pi  
angle_in_rad    = value_of_pi*angle_in_degree/180.0  
var2            = sin(angle_in_rad)  
  
print("sin(", angle_in_degree,") is ", var2)
```

Directly access ***pi*** or ***sin***

```
# ----- Module import variation 4 -----  
from math import pi, sin, cos  
  
# variables initialization  
angle_in_degree = 45  
value_of_pi     = pi  
angle_in_rad    = value_of_pi*angle_in_degree/180.0  
var2            = sin(angle_in_rad)  
  
print("sin(", angle_in_degree,") is ", var2)
```

Directly access ***pi*** ***sin*** and ***cos*** (in a single import line) but nothing else

<https://docs.python.org/3/tutorial/modules.html>

# Random number generation

- Steps for generating a random number are as follows:
  - Step 1: Import the **random** module
  - Step 2: Generate a random number (eg, an integer number) between a range of values denoted by a lower\_range and an upper\_range
    - For example, in order to generate a random integer between lower\_range of 1 and upper\_range of 10, we need to do the following:

```
import random  
  
rand_number = random.randint(1, 10)  
print(rand_number)
```

# Local and global variables

- Local variables:
  - Variables declared 1) inside function 2) function parameters
  - Only visible to the defined function
- Global variables:
  - Variables that are defined outside of user defined functions
  - Can be accessed by any function after creation
  - Global variable can be replaced/hidden by local variable if declared with the same name

# Scope: local and global variables

- Global variables:

- Variables that are defined outside of user defined functions
- Can be accessed by any function after creation
- Global variable can be replaced/hidden by local variable if declared with the same name

```
num1 = 1

# defining user defined functions
def dummy_function1():
    num1 = 2
    print("Inside function dummy_function1: num1 is local variable ", num1)

|
print("Before calling dummy_function1() value of num1 = ", num1)

dummy_function1()

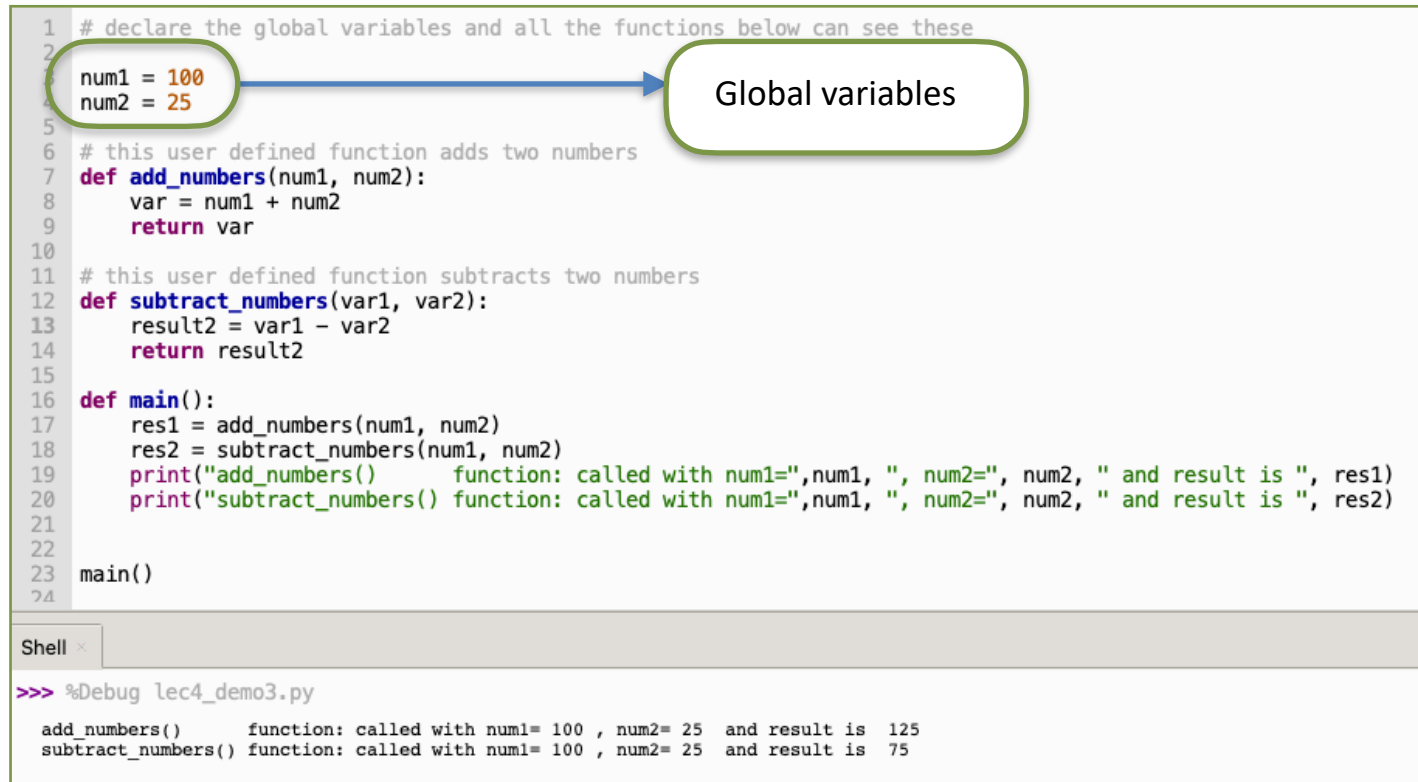
print("After calling dummy_function1() value of num1 = ", num1)
```

```
Before calling dummy_function1() value of num1 = 1
Inside function dummy_function1: num1 is local variable 2
After calling dummy_function1() value of num1 = 1
```

# Scope: local and global variables

- Global variables:

- Variables that are defined outside of user defined functions
- Can be accessed by any function
- Here values of global variables are copied to the parameters during function call



The screenshot shows a Python script in a text editor. Lines 1-2 are comments. Lines 3-4 define global variables `num1 = 100` and `num2 = 25`. A blue arrow points from these lines to a callout box labeled "Global variables". Lines 6-9 define a function `add_numbers` that takes `num1` and `num2` as arguments and returns their sum. Lines 11-14 define a function `subtract_numbers` that takes `var1` and `var2` as arguments and returns their difference. Lines 16-24 define a `main` function that calls both `add_numbers` and `subtract_numbers` with the global variables and prints the results. Below the editor is a "Shell" window showing the execution output, which matches the print statements in the `main` function.

```
1 # declare the global variables and all the functions below can see these
2
3 num1 = 100
4 num2 = 25
5
6 # this user defined function adds two numbers
7 def add_numbers(num1, num2):
8     var = num1 + num2
9     return var
10
11 # this user defined function subtracts two numbers
12 def subtract_numbers(var1, var2):
13     result2 = var1 - var2
14     return result2
15
16 def main():
17     res1 = add_numbers(num1, num2)
18     res2 = subtract_numbers(num1, num2)
19     print("add_numbers()      function: called with num1=", num1, ", num2=", num2, " and result is ", res1)
20     print("subtract_numbers() function: called with num1=", num1, ", num2=", num2, " and result is ", res2)
21
22
23 main()
24
```

Shell

```
>>> %Debug lec4_demo3.py
add_numbers()      function: called with num1= 100 , num2= 25 and result is  125
subtract_numbers() function: called with num1= 100 , num2= 25 and result is  75
```



# Scope: local and global variables

- Scope resolution: Mechanism of searching for a name, e.g., variable or function

- **Step 1:** search the referenced name in the local scope. If not found, then go to step 2
- **Step 2:** search the referenced name in the global scope. If not found, then go to step 3
- **Step 3:** If searched name is not found in either step 1 or step 2, then search in the built-in scope
- **Step 4:** If not found in the above steps, then interpreter generates an Error message

# Global variables

- Global variables are defined outside of user defined functions or they can be introduced by the **global** statement
- As you have noticed by now, they can be source of confusion
  - Name clashing
  - Order of their definitions matter
- Use of global variables is not recommended, better to avoid or at least minimize their usage
- If you need to use eg, some constants, then declare them using capital letters

```
VALUE_OF_PI = 3.14  
MILES_TO_KILOMETERS = 1.619
```

# Topics

- Variables, expression
  - Functions
  - Scope for local and global variables
- Boolean type and boolean expression
  - Selection statements are useful for branching inside your program
    - if
    - if-else
    - if-elif-...-else
- Sequence
    - String
    - List
  - The **while** loop
  - The **for** loop to solve a repetitive task
    - Value **for** loop
    - Index **for** loop
    - Nested **for** loop

# ‘Bool’ Data Type

- Notion of something being true and being false — represented with two ‘bool’ data types:
  - True
  - False
- Allows us to evaluate true or false questions — in real life, we always encounter question with Yes or No answer
- Logical and comparison operators:
  - Boolean expression with logical operator (and, or, not)
  - Boolean expression with comparison operator (<, <=, >, ==, etc)

# Boolean Expression

- Expressions that are evaluates to two '**bool**' types
- Operations with logical operators — **and/or/not**
  - **and** – given two boolean, are both True? answer is True  
**boolean expression<sub>1</sub> and boolean expression<sub>2</sub>**
  - **or** – given two booleans, at least one is True? answer is True  
**boolean expression<sub>1</sub> or boolean expression<sub>2</sub>**
  - **not** – given a boolean expression, switch between True/False  
**not boolean expression**

# Logical Operators

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

- expression<sub>1</sub> and expression<sub>2</sub>

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

expression<sub>1</sub> or expression<sub>2</sub>

x	not x
False	True
True	False

not expression

# Comparison Operators

- We can write expression that evaluates to boolean with other comparison operators
  - Compare two values or check something

Description	Example	Result
Less than	$2 < 15$	True
Greater than	$2 > 15$	False
Less than or equal	$2 \leq 15$	True
Greater than or equal	$2 \geq 15$	False
Equality check	$2 == 15$	False
Inequality check	$2 \neq 15$	True

# More Boolean Expressions

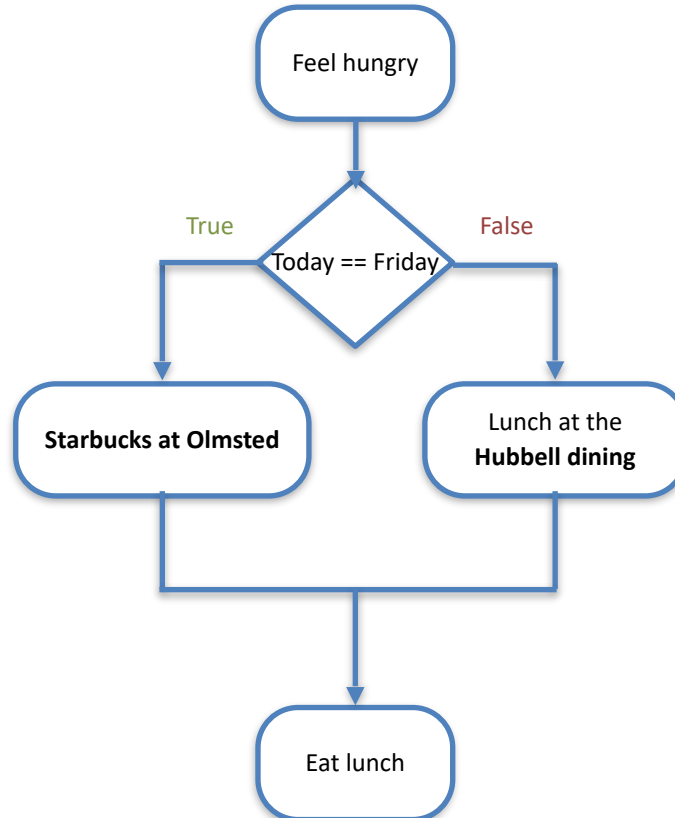
X	Y	X and Y
2 < 15	2 >=15	False
3 < 15	2 ==15	False
3 < 15	15 == 15	True
16 > 15	2 != 15	True

- expression<sub>1</sub> and expression<sub>2</sub>



# Selection Statements

- Program taking one *path* or *branch* of the code instead of taking another, based on the **boolean expression**'s value
- This feature allows to ask true/false questions in the code. Depending on the boolean answer (True or False), the program will execute a specific branch



# ‘if’ statement

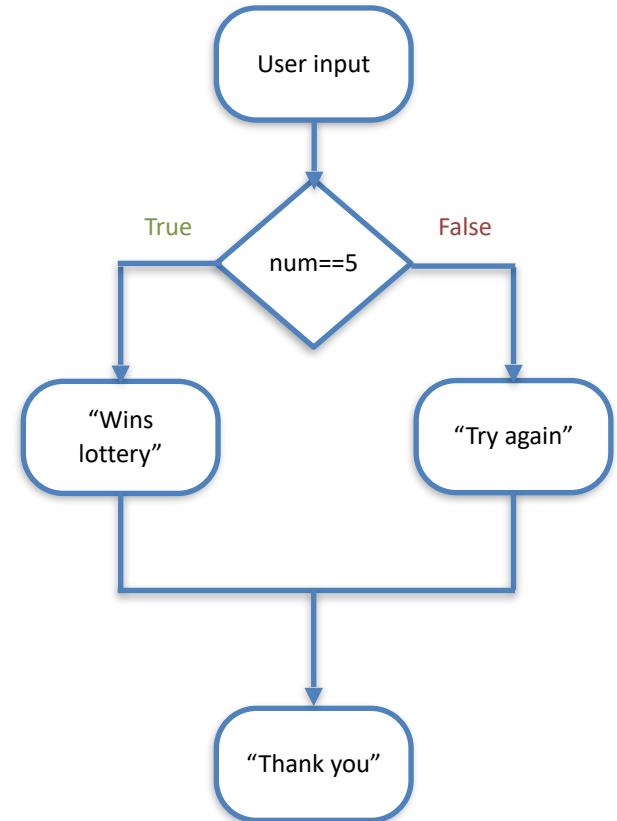
- **if** <condition expression> :

    <block statements>

- **condition expression**: a boolean expression
- **block statements**: statements to be executed if result of the condition expression is **True**
- Notice: indention is required to define a **block statements** and also notice a colon at the end of the condition expression

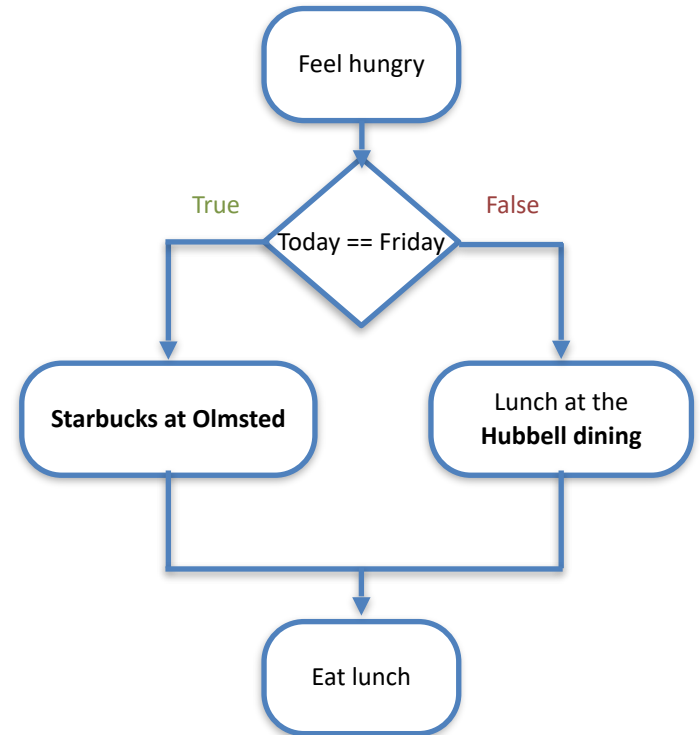
# 'if ... else' Statement

```
1 num = int(input("Please enter a number. "))
2 if num == 5:
3     print("Yeah! I won a lottery ...")
4 else:
5     print("Oh gosh! better luck next time ...")
6 print("Thank you!")
7
```



# Multiple Selections

- We may need to branch in more than two directions — multiple selection
  - Can have nested if-statement
  - Keyword **elif** (short for ‘else if’) introduces a new structure
  - Blocks with multiple **elif** conditions structures are referred to as mutually exclusive structures.

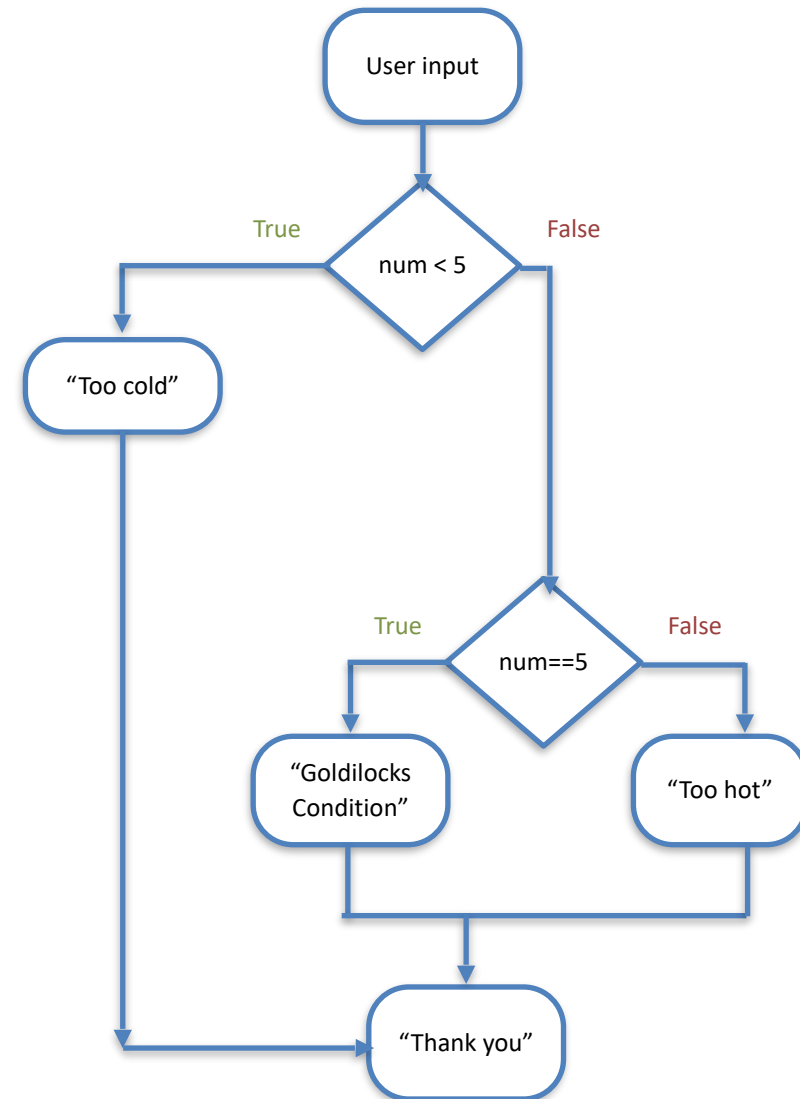


# Multiple Selections

```
1 num = int(input("Please enter a number. "))
2 if num < 5:
3     print("Too cold ...")
4 elif num == 5:
5     print("Perfect! Goldilocks condition ...")
6 else:
7     print("Too hot ...")
8 print("Thank you!")
9
```

Shell x

```
>>> %Run test4.py
Please enter a number. 5
Perfect! Goldilocks condition ...
Thank you!
>>> |
```



# Super Important: Multiple Selections

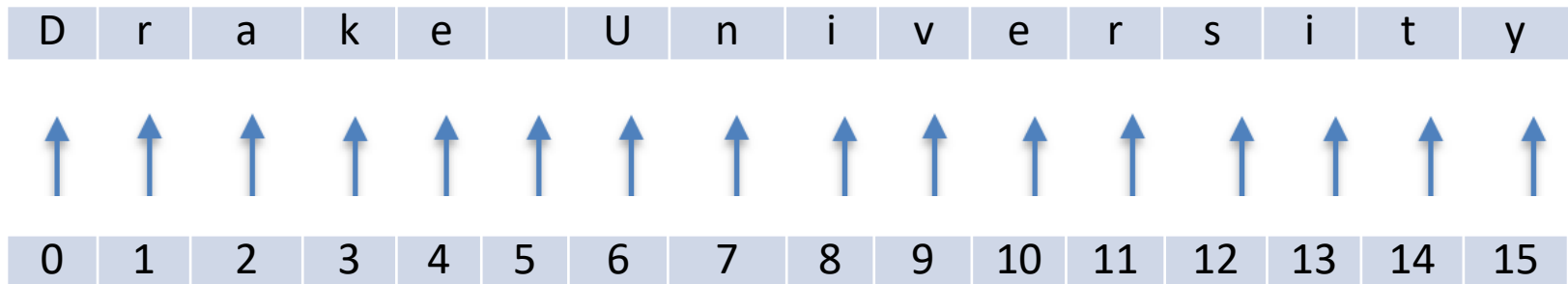
- We can have multiple if blocks but they are not disconnected
- We can have multiple nested if-elif-else blocks

# Topics

- Variables, expression
  - Functions
  - Scope for local and global variables
- Boolean type and boolean expression
  - Selection statements are useful for branching inside your program
    - if
    - if-else
    - if-elif-...-else
- Sequence
    - String
    - List
  - The **while** loop
  - The **for** loop to solve a repetitive task
    - Value **for** loop
    - Index **for** loop
    - Nested **for** loop

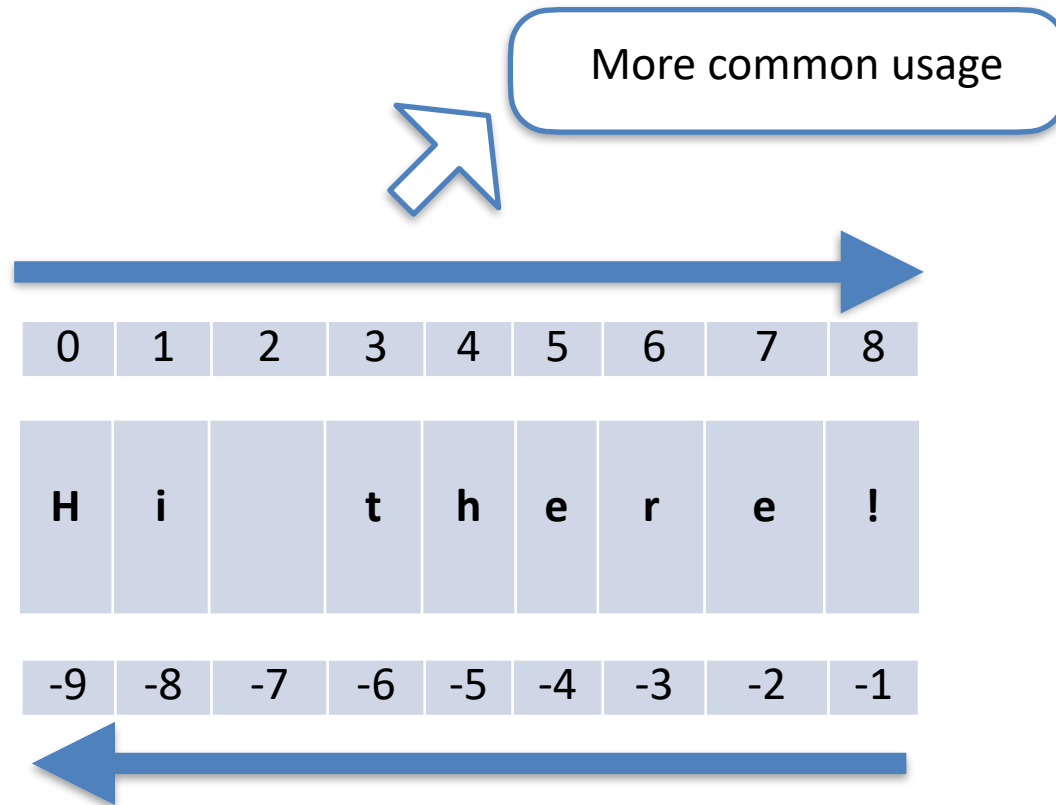
# Sequence: Strings

- Sequence is an ordered group of elements (numbers, characters, etc)
- String is a sequence of characters
  - “Drake University”
  - “cs65:introduction\_to\_computer\_science!”
- Each position in a sequence is marked with an **index** or **position**
  - Starts (from left) at position  $0$  and ends at position  $(length-1)$
  - Start indexing from the *left to right*





# Summary: Indexing

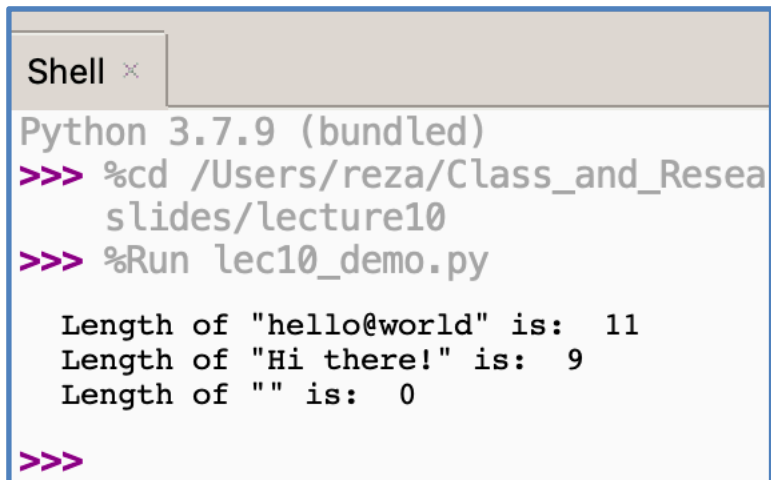


# Length of a Sequence

- How can you find the length of a string?
  - Use built-in *len()* function

```
my_string1 = "hello@world"
my_string2 = "Hi there!"
my_string3 = ""

print("Length of \"hello@world\" is: ", len(my_string1))
print("Length of \"Hi there!\" is: ", len(my_string2))
print("Length of \"\" is: ", len(my_string3))
```



The screenshot shows a terminal window titled "Shell" with a close button. It displays the execution of a Python script using the bundled Python 3.7.9. The user navigates to the directory /Users/reza/Class\_and\_Research/slides/lecture10 and runs the file lec10\_demo.py. The script's output is printed to the terminal, showing the lengths of the three strings defined in the code above: 11 for "hello@world", 9 for "Hi there!", and 0 for an empty string. The prompt >>> is visible at the bottom of the terminal.

```
Shell x
Python 3.7.9 (bundled)
>>> %cd /Users/reza/Class_and_Research/slides/lecture10
>>> %Run lec10_demo.py

Length of "hello@world" is:  11
Length of "Hi there!" is:   9
Length of "" is:  0

>>>
```

# Accessing items with index

- Use *variable\_name[index]* access an item in a sequence

```
15 # -----
16 # demo 2 accessing elements in a string
17 my_string1 = "Drake University"
18 my_string2 = "Hi there!"
19
20 vis = 1
21 if (vis):
22     print("Character at index = 0 is ", my_string1[0])
23     print("Character at index = 1 is ", my_string1[1])
24     print("Character at index = 2 is ", my_string1[2])
25     print("Character at index = 15 is ", my_string1[15])
26
27
```

Shell x

```
>> %Run lec10_demo.py
```

```
Character at index = 0 is D
Character at index = 1 is r
Character at index = 2 is a
Character at index = 15 is y
```

# Sequence: List

- Sequence is an ordered group of elements (numbers, characters, etc)
- **String** is a type of sequence whose members are characters
  - “Drake University”
  - “cs65:introduction\_to\_computer\_science!”
- **List** is another type of sequence whose members can be numbers, strings, or even another list!
  - [“Drake University”, “hello”, “world”]
  - [1, 2, 3, 4, 5]
  - List will be discussed in greater detail in a separate lecture

# Random Number

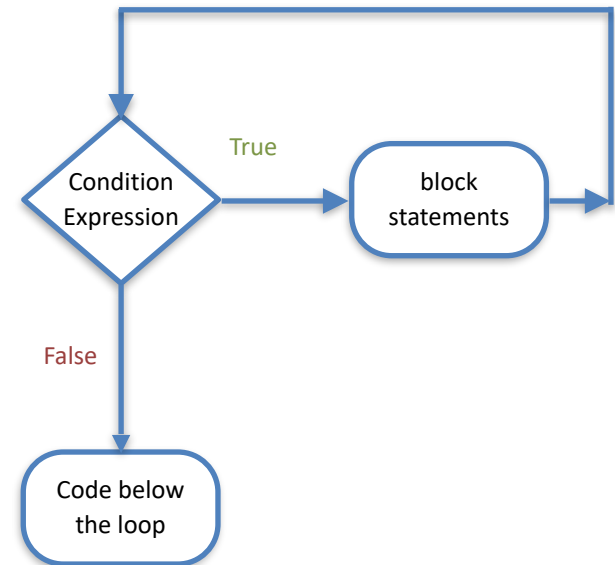
- Random numbers are useful several programming tasks:
  - Simulating a coin toss — random flipping of head or tail
  - Simulating a dice roll — random roll of one of six sides
  - Simulating a card shuffling from 52 cards
- Python provides library to generate random numbers
  - Like math module or graphics module, you can import random module to get access to random number generating functions

# Syntax for **while** Loop

- **while** <condition expression> :

<block statements>

- **condition expression**: a boolean expression
- **block statements**: statements to be executed if result of the condition expression is **True**
- Unlike if statement, the <**block statements**> will repeatedly be executed until the <**condition expression**> becomes False



# The while Loop

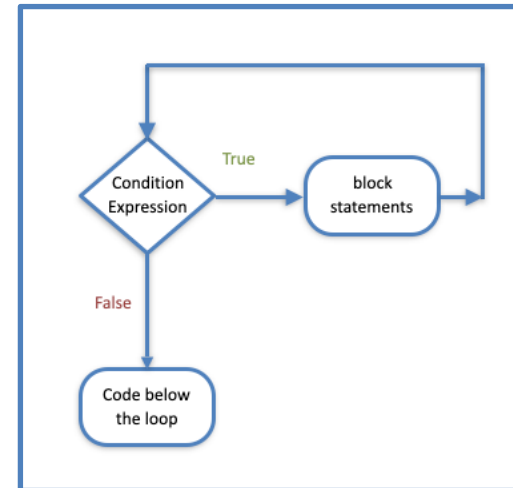
- Section of code that repeats — designed to solve a repetitive task
  - decrease the value of a variable by 1 until it becomes negative

```
num = 5

while num > 0:

    print(num)
    num = num - 1

>>> %Run lecture8_while.py
5
4
3
2
1
>>>
```

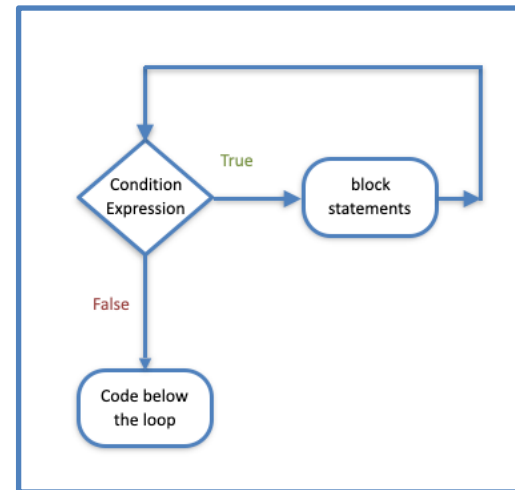


# The while Loop

- **Infinite loop**: section of code that repeats forever
  - The condition expression should be designed carefully so that the loop terminates after a certain number of iterations

```
num = 5
while num > 0:
    print(num)
    num = num + 1
```

What will happen?





# The while Loop

- The index variable can be updated (**decreased**) with a shorthand:

```
num = 5

while num > 0:

    print(num)
    num = num - 1

>>> %Run lecture8_while.py
5
4
3
2
1
>>>
```

```
num = 5

while num > 0:

    print(num)
    num -= 1

>>> %Run lecture8_while.py
5
4
3
2
1
>>>
```

# The while Loop

- The index variable can be updated (**increased**) with a shorthand:

```
num = 5
while num > 0:
    print(num)
    num = num + 1
```

```
num = 5
while num > 0:
    print(num)
    num += 1
```

# Syntax for value for loop

- **for** variable **in** [1, 2, ..., 5] :  
    **statements**
- Statements will be repeated sequentially from first to last item in a sequence (here it will be repeated 5 times since there are 5 numbers in the List)
  - Iteration 1: variable will be assigned **1**
  - Iteration 2: variable will be assigned **2**
  - ...
  - Iteration 5: variable will be assigned **5**

# Summary: value for loop

```
for var in [1, 2, 3, 4, 5]:  
    new_var = var*10  
    print("10 times", var, " is: ", new_var)
```

```
>>> %Run lec10_demo.py
```

```
10 times 1 is: 10  
10 times 2 is: 20  
10 times 3 is: 30  
10 times 4 is: 40  
10 times 5 is: 50
```

# Summary: value for loop visualization

```
for var in [12, 13, 14, 15, 16]:  
    print("current num is: ", var)
```

Empty

variable

Full

	12	13	14	15	16
--	----	----	----	----	----

12	←	12	13	14	15	16
----	---	----	----	----	----	----

13	←		13	14	15	16
----	---	--	----	----	----	----

14	←			14	15	16
----	---	--	--	----	----	----

15	←				15	16
----	---	--	--	--	----	----

16	←					16
----	---	--	--	--	--	----

with a value

Empty

# Summary: *range()* function

- The *range()* function simplifies the process of for loop writing
- Creates a sequence of numbers on the fly
- These numbers can be used to index the sequence

```
# version 1:
print("range() function version 1:")
for var in range(5):
    print(var)

# version 2: start, stop
print("range() function version 2:")
for var in range(0, 5):
    print(var)

# version 3: start, stop, step_size
print("range() function version 3:")
for var in range(0, 10, 2):
    print(var)
```

# Value for loop vs Index for loop

- So far we have seen the syntax of **value for loop**

```
for var in [10, 20, 30, 40, 50] :  
    print(var)
```

- There is another form called **index for loop**

```
my_list = [10, 20, 30, 40, 50]  
length = len(my_list)  
for i in range(length) :  
    print( my_list[i] )
```

common practice is to name the  
index variables with **i**, **j**, or **k**

# Try to finish the exercises shown in class

- Write a loop that will print '\*' 5 times.
- Write a loop that will print '\*' 10 times.
- Write a loop that will print '\*' N times (prompt the user to enter this number)
- Find the sum of all the numbers from 1 to max\_num
  - eg,  $1 + 2 + 3 + 4 + 5 = 15$
  - use **for** loop to do this
- Find the average of these numbers



# Try to finish the exercises shown in class

- Finding a number (prompt the user to enter that number) in a given list of number

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

```
# ----- finding a number in a list -----  
my_list = [1, 3, 5, 7, 9, 11]  
cur_num = int(input("enter the number you are looking for in the list: "))  
flag_found = False  
for val in my_list:  
    if (val == cur_num):  
        flag_found = True  
  
if (flag_found):  
    print("Found ", cur_num, "! Yay!")  
else:  
    print("Could not find ", cur_num, " in the list :'(")
```

# Try to finish the exercises shown in class

- Counting how many times a number (prompt the user to enter that number) appears in a given list.

```
my_list = [1, 1, 1, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, 5, 5, 7]
```

```
my_list = [1, 1, 1, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, 5, 5, 7]
cur_num = int(input("enter the number for which you want find the count: "))
count = 0
for val in my_list:
    if (cur_num == val):
        count = count + 1
if (count > 0):
    print("Your number ", cur_num, " appears ", count, " times in the list.")
else:
    print("Could not find your number ", cur_num, " in the list.")
```

# Try to finish the exercises shown in class

- Finding the **location** of given a number (prompt the user to enter that number) in a given list.

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

- Finding the **maximum** number in a given list.
- Finding the **maximum** number in a given list.

# Nested for loops

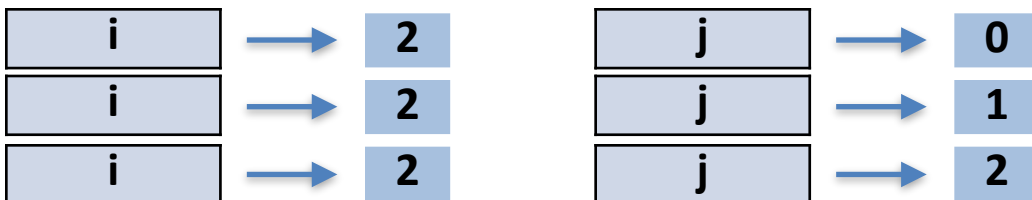
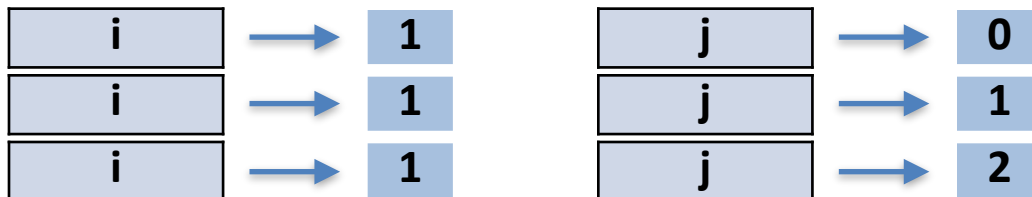
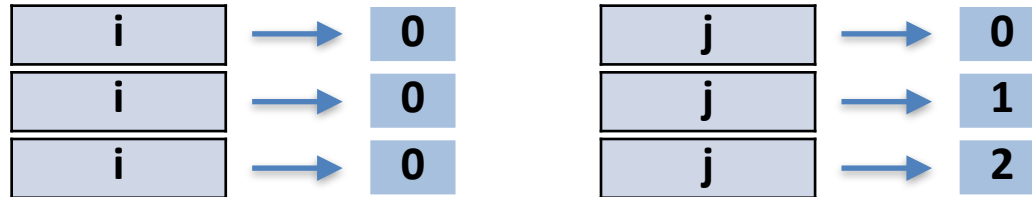
- Putting one loop inside another
  - The first loop is called the outer loop
  - The second loop is called the inner loop

```
for i in range(3):  
    for j in range(3):  
        print("i: ", i, "j: ", j)
```

# Visualization of nested **for** loop



```
# nested for loop
for i in range(3):
    print("Enters outer loop")
    for j in range(3):
        print("\tInner: i ->", i, " j ->", j)
```



# Thonny output: nested for loop

```
# nested for loop
for i in range(3):
    print("Enters outer loop")
    for j in range(3):
        print("\tInner: i ->", i, " j ->", j)
```

```
>>> %Run lec10_demo.py

Enters outer loop
    Inner: i -> 0  j -> 0
    Inner: i -> 0  j -> 1
    Inner: i -> 0  j -> 2
Enters outer loop
    Inner: i -> 1  j -> 0
    Inner: i -> 1  j -> 1
    Inner: i -> 1  j -> 2
Enters outer loop
    Inner: i -> 2  j -> 0
    Inner: i -> 2  j -> 1
    Inner: i -> 2  j -> 2
```

# Comments

- Comments are notes explaining the functionality of your computer program (source code)
- Python comments are denoted with
  - `#` for a single line
  - triple quotes (either `'` or `"`) for multiple lines
- Other languages eg, C++ has different syntax

```
# Author's name: Md Alimoor Reza
# Author's contact: md.reza@drake.edu
# Date: (September 1st, 2021)
# Collaborator:
#   Your partner's name

#print("Yay! this is my first python program in CS65!")
```

```
"""
Author's name: Md Alimoor Reza
Author's contact: md.reza@drake.edu
Date: (September 1st, 2021)
Collaborator:
    Your partner's name

#print("Yay! this is my first python program in CS65!")"""
```