CSCI-UA-0002

Intro to Computer Programming (No Prior Experience)

Midterm Review

Professor Emily Zhao

Section 008
T/R 12:30-1:45PM

Section 012
T/R 4:55-6:10PM

Agenda

- Starred Topics
- Open Questions
- Practice Problems
- Assignment 6 Workshop

Starred Topics

- Boolean Variables
- Comparing Strings
- Nested Conditionals
- Escape Characters
- Dynamic Formatting
- For Loops
  - User controlled-ranges
  - Interacting through a list, string, range
- Functions
  - What are they?
  - When do we use them?

Boolean Variables

- Essential for managing program flow, making decisions, and controlling loops
- They help in creating dynamic and flexible programs by allowing you to switch between different branches of code based on the truth or falsity of the variable's value

Conditional Statements

```
is_raining = True
if is_raining:
    print("Don't forget your umbrella!")
```

Loops

```
keep_running = True
while keep_running:
    user_input = input("Do you want to continue? (yes/no): ")
    if user_input.lower() == "no":
        keep_running = False
```

Data Validation

```
is_valid_input = False
while not is_valid_input:
    user_input = input("Enter a number between 1 and 10: ")
    if 1 <= int(user_input) <= 10:
        is_valid_input = True
    else:
        print("Invalid input. Try again.")
```

Flagging State

```
is_light_on = False
```

Toggling Features

```
feature_enabled = True
```

Function Return Values

```
def is_even(number):
    return number % 2 == 0
```

Error Handling

```
error_occurred = False
try:
    # Code that may raise an exception
except Exception as e:
    error_occurred = True
    print(f"An error occurred: {e}")
```

Nested Conditionals

- Allow you to handle complex decision-making scenarios in your code by considering multiple conditions and their combinations.
- However, be cautious not to overcomplicate your code with excessive nesting, as it can become difficult to read and maintain.
- In such cases, consider using elif clauses for better code organization

Nested Conditionals

```
if condition1:

    # Code to execute when
    # condition1 is True

    if condition2:
        # Code to execute when both
        # condition1 and condition2
        # are True
    else:
        # Code to execute when
        # condition1 is True,
        # but condition2 is False
else:
    # Code to execute when
    # condition1 is False
```

```
age = 25
is_student = True

if age >= 18:
    if is_student:
        print("You are an adult student.")
    else:
        print("You are an adult.")
else:
    print("You are a minor.")
```

Comparing Strings

- When you use comparison operators like <, >, <=, and >= to compare strings, Python performs lexicographic (alphabetical) comparisons.
- For each pair of characters in the strings being compared, Python compares their ASCII values to determine their order in the alphabet. The comparison stops at the first character that differs between the strings.
- String comparisons are case-sensitive, meaning uppercase and lowercase letters are treated as distinct characters with different ASCII values. You can use the `.lower()` or `.upper()` methods to convert the strings to the same case before comparing them.

Standard ASCII Table

0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Comparing Strings

```
string1 = "Apple"
string2 = "apple"

if string1.lower() == string2.lower():
    print("The strings are equal (case-insensitive).")
```

What's the output?

```
"dog" > "cat"          → True

"Camel" < "camel"      → True

"dog" < "dogfight"     → True
```

* **For practice today, please exclusively use the code editor allowed on the midterm:**

  https://prager.hosting.nyu.edu

Midterm Programming Challenge #1 (on Ed)

Goals:

- Practice importing midterm module
- Practice using the code editor

Topics:

- Nested Conditionals
- String Comparison
- Functions

Escape Characters

- An “escape character” allows you to perform special actions inside the confines of a delimiter
- In Python, the escape character is `\`
- It causes Python to treat the next character as “special”

```
print('Hi, I\'m Harry Potter, a wizard.')
```

Escape Characters

- There are a number of special characters you can use in conjunction with the escape character to perform special string operations
- `\n` forces a line break
- `\t` creates a tab

```
print ("line 1\n\tline 2\nline 3\n")

# line 1
#    line 2
# line 3
```

Line Continuation

- Sometimes the code you write can get very long
- You can use the `\` symbol to indicate to Python that you would like to continue your code onto another line

```
1 print("Once upon a time, there was a king; who used to wear a single \
2         horned crown. He had a lavish palace, three beautiful wives, \
3         and seven children; all well qualified in their respective fields. \
4         The king was reaching the retirement age, so he asked his elder son \
5         to lead his empire so that he could undergo seclusion.")
```

Dynamic Formatting (for Tables)

1. Make all the widths of your cells the same keeping in mind the maximum width you could encounter
2. Create formatting specs with that width, add necessary buffer spaces
3. Align individual cells accordingly and provide the correct endings based on data type (s, d, f)

```
# Problem: I want all names to be right aligned and 15 spaces wide

# Expected Output:
#           Mabel
#            Greg
#             Min
#           Peter

name1 = "Mabel"
name2 = "Greg"
name3 = "Min"
name4 = "Peter"

# Here's how I would hardcode it
print(format(name1, ">15s"))
print(format(name2, ">15s"))
print(format(name3, ">15s"))
print(format(name4, ">15s"))

# How would I implement a variable into my formatting spec instead?
width = 15

# This won't work:
print(format(name1, ">widths"))
```

```
    print(format(name1, ">widths"))
ValueError: Invalid format specifier
```

```
# The answer: concatenation!
# The formatting spec is a string

# Just as we would recreate the following string "October28":
month = "October"
date = 28
date_string = month + str(28)

# We can do the same thing with our formatting spec
width = 15
formatting_spec = ">" + str(width) + "s"

print(format(name1, formatting_spec))
print(format(name2, ">" + str(width) + "s"))

# ^ Both work!
# Output:
#          Mabel
#           Greg
```

Programming Challenge: Stair Steps

Write a program using a for loop that prints out the following pattern of characters. (If you're stuck, print the output first without for loops. Is there a pattern?)

```
**
****
******
********
**********
************
```

Extension: ask the user for starting starting amount and ending amount of asterisks, making sure that the ending amount is greater than the starting amount. You can also ask the user for the "step" value at which they would like to increase the asterisks by.

Programming Challenge: Stair Steps

Extension: Right align the output

Programming Challenge: Stair Steps [Solution pt. 1]

```
**

****

******

********

**********

************
```

What's the pattern?

```
# hard-code it first

print("*" * 2)
print("*" * 4)
print("*" * 6)

# looks like I need to generate 2, 4, 6...
# I can do that with range
# start at 2, end at 12+1, step by 2
# range(2, 12+1, 2)

for i in range(2, 13, 2):
    print("*" * i)
```

Programming Challenge: Stair Steps [Solution pt. 2]

```
'''
# previous solution
for i in range(2, 13, 2):
    print("*" * i)
'''

# to take in user input:
# need to replace 2 with input1
# need to replace 13 with input2 + 1
# replace 2 with user's step

start = int(input("Starting number of *s: "))
end = int(input("Ending number of *s: "))
step = int(input("Step increment: "))

for i in range(start, end+1, step):
    print("*" * i)
```

```
# user validation
while True:
    start = int(input("Starting number of *s: "))
    if start <= 0: # run loop again if start is not positive
        print("Invalid input. Try again.")
    else:
        break
while True:
    end = int(input("Ending number of *s: "))
    if end <= start: # end must be greater than start
        print("End number needs to be greater than start.")
    else:
        break

while True:
    step = int(input("Step increment: "))
    if step <= 0: # step must be 1 or more
        print("Step must be greater than or equal to 1.")
    else:
        break

for i in range(start, end+1, step):
    print("*" * i)
```

Programming Challenge: Stair Steps [Solution pt. 3]

```
# user validation
while True:
    start = int(input("Starting number of *s: "))
    if start <= 0: # run loop again if start is not positive
        print("Invalid input. Try again.")
    else:
        break
while True:
    end = int(input("Ending number of *s: "))
    if end <= start: # end must be greater than start
        print("End number needs to be greater than start.")
    else:
        break
while True:
    step = int(input("Step increment: "))
    if step <= 0: # step must be 1 or more
        print("Step must be greater than or equal to 1.")
    else:
        break

# how long does each line need to be?
# needs to be length of the longest line
# use end to create formatting string:
formatting_spec = ">" + str(end) + "s"

for i in range(start, end+1, step):
    print(format("*" * i, formatting_spec))
```

Iterables

- An iterable is any Python object capable of returning its elements one at a time.
- Iterable types that we've been working with so far are strings, range objects, and lists.

```
# STRING
# a string returns its charcters
# one at a time

for char in "Emily":
    print(char)

# > E
# > M
# > I
# > L
# > Y
```

Iterables

- An iterable is any Python object capable of returning its elements one at a time.
- Iterable types that we've been working with so far are strings, range objects, and lists.

```
# LIST
# a list returns its
# (comma separated) elements
# one at a time

for name in ["Emily", "Greg", "Mabel"]:
    print(name)

# > Emily
# > Greg
# > Mabel
```

Iterables

- An iterable is any Python object capable of returning its elements one at a time.
- Iterable types that we've been working with so far are strings, range objects, and lists.

```
# RANGE
# a range returns its
# range of numbers
# one at a time

for num in range(0, 5, 2):
    print(num)

# > 0
# > 2
# > 4
```

Nested For Loops

- The outer loop iterates over an outer iterable
- The inner loop iterates over an inner iterable
- For each iteration of the outer loop, the inner loop completes all its iterations.

```
for char in "HI": # H, I
    for num in range(0,5,2): # 0, 2, 4
        print(char, num)

# H 0
# H 2
# H 4
# I 0
# I 2
# I 4
```

Why Functions?

- **Modularity and Reusability:** Functions allow you to break down a complex program into smaller, manageable, and reusable parts. You can write a function to perform a specific task or solve a particular problem, and then you can reuse that function whenever you need that task performed.
- **Abstraction:** When you call a function, you don't need to know how it's implemented; you only need to know what it does. This separation of concerns makes code easier to understand and maintain.
- **Code Organization:** Functions help you organize your code logically. Instead of having all your code in a single block, you can group related code into functions.

Why Functions?

- **Readability:** Functions make code more readable by giving meaningful names to blocks of code. A well-named function tells you what it does, reducing the need for extensive comments.
- **Return values:** You can use these return values for further processing
- **Code Sharing:** Functions can be shared as libraries or modules, making them accessible to other programmers
- **Performance:** you can optimize specific functions independently without affecting the rest of the program
- **Scoping:** prevents naming conflicts and allows you to reuse variable names

```python
print("Geometry Area Calculator")
print("1. Calculate Square Area")
print("2. Calculate Rectangle Area")
print("3. Calculate Circle Area")

choice = input("Enter your choice (1/2/3): ")

if choice == '1':
    side_length = float(input("Enter the side length of the square: "))
    area = side_length * side_length
    print("The area of the square is: ", area)
elif choice == '2':
    length = float(input("Enter the length of the rectangle: "))
    width = float(input("Enter the width of the rectangle: "))
    area = length * width
    print("The area of the rectangle is: ", area)
elif choice == '3':
    radius = float(input("Enter the radius of the circle: "))
    pi = 3.14159
    area= pi * radius * radius
    print("The area of the circle is: ", area)
else:
    print("Invalid choice. Please select 1, 2, or 3.")
```

```python
# Function to calculate the area of a square
def calculate_square_area(side_length):
    return side_length * side_length

# Function to calculate the area of a rectangle
def calculate_rectangle_area(length, width):
    return length * width

# Function to calculate the area of a circle
def calculate_circle_area(radius):
    pi = 3.14159
    return pi * radius * radius

# Main program
def main():
    print("Geometry Area Calculator")
    print("1. Calculate Square Area")
    print("2. Calculate Rectangle Area")
    print("3. Calculate Circle Area")

    choice = input("Enter your choice (1/2/3): ")

    if choice == '1':
        side_length = float(input("Enter the side length of the square: "))
        area = calculate_square_area(side_length)
        print("The area of the square is: ", area)
    elif choice == '2':
        length = float(input("Enter the length of the rectangle: "))
        width = float(input("Enter the width of the rectangle: "))
        area = calculate_rectangle_area(length, width)
        print("The area of the rectangle is: ", area)
    elif choice == '3':
        radius = float(input("Enter the radius of the circle: "))
        area = calculate_circle_area(radius)
        print("The area of the circle is: ", area)
    else:
        print("Invalid choice. Please select 1, 2, or 3.")

main()
```

Programming Challenge: Factorials

Write a program that calculates the factorial of a number given by the user. The factorial of a number is the product of all positive integers less than or equal to the number. Factorial is indicated by a number followed by an exclamation point. For example, the factorial of 4 is: 4! = 4 x 3 x 2 x 1 = 24. **(Extension**: Turn this program into a function that returns the factorial string)

```
Run Example #1
Calculate the factorial of: 6
6! = 6 * 5 * 4 * 3 * 2 * 1 = 720

Run Example #2
Calculate the factorial of: 0
0! = 1

Run Example #3
Calculate the factorial of: -2
Number must be >= 0
```

Homework

- Midterm Prep Quiz
- Study for Midterm (this Thursday)
- Assignment #6 (due this Thursday)