香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

AIE 1001 Introduction to AI Programming

Midterm Review

Prof. Junjie Hu

School of Artificial Intelligence

Part of the slides are adapted from NJU SICP

Scope & Reminders

- Lecture **1** – Lecture **6** (Recursion)
- Question:
  - Single Choice (one correct option, 4*10, 40 points), Multiple Choice (one or more correct options, 5*3, 15 points)
  - Short answer question (20+25, 45 points)
    - Including, but not limited to: Code Fault Localization, Code Completion, Function Design
- Exam Duration: 1 hr
- Remember to bring **your Student ID card,** and write your student ID number to the answer sheet.
- No cheat sheet, calculator, digital devices allowed

1. Data Representation

- **Positional Notation:** Understanding the concept of a base and a set of symbols for representing numbers.
- **Number Systems:**
  - Binary (base 2)
  - Octal (base 8)
  - Decimal (base 10)
  - Hexadecimal (base 16)

- **Number System Conversion:**
  - Converting from any number system (Binary, Octal, Hexadecimal) to the Decimal system.
  - Converting Decimal numbers to Binary numbers, including both integer and fractional parts.
  - Converting between Binary and Octal by grouping three digits.
  - Converting between Binary and Hexadecimal by grouping four digits.

Decimal number system

- In the decimal number system, the base is 10, the symbols include 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- Every number can be decomposed into the sum of a series of numbers, each is represented by a positional value times a weight

- $N = a_n \times 10^n + a_{n-1} \times 10^{n-1} + a_{n-2} \times 10^{n-2} \ \ldots\ldots + a_0 \times 10^0 + a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} \ldots$

- a_n is the positional value (ranging from 0 to 9), while 10^n represents the weight

Binary number system

- In the binary system, the base is 2, we use only two symbols 0 and 1
- "10" is used when we hit 2 (逢二进一)
- $N = a_n \times 2^n + a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} \ \ldots\ldots + a_0 \times 2^0 + a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} \ldots$

$$9_{(10)} = 1\times 2^3 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0 = 1001_{(2)}$$

- a_n is the positional value (ranging from 0 to 1), while 2^n represents the weight

Hexadecimal number system

- In the hexadecimal system, the base is 16, we use 16 symbols {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f}

- “10” is used when we hit 16 (逢十六进一)

- $N = a_n \times 16^n + a_{n-1} \times 16^{n-1} + a_{n-2} \times 16^{n-2} \ \ldots\ldots + a_0 \times 16^0 + a_{-1} \times 16^{-1} + a_{-2} \times 16^{-2} \ldots$

- a_n is the positional value (ranging from 0 to 15), while 16^n represents the weight

Notes:

Binary number <-> Hexadecimal number

<-> octal number

<-> decimal number

All the representations can be converted into each other

Converting binary number into decimal number

Example $(1101.01)_2$

$$=(1\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0+0\times 2^{-1}+1\times 2^{-2})_{10}$$

$$=(13.25)_{10}$$

Practice $(10110.11)_2 = (?)_{10}$

Converting octal number into decimal number

Example $(24.67)_8=(2 \times 8^1+ 4 \times 8^0+6 \times 8^{-1}+7 \times 8^{-2})_{10}$

$=(20.859375)_{10}$

Practice $(35.7)_8= (?)_{10}$

Converting decimal fraction into binary fraction

Example: $(0.875)_{10} = (?)_2$

Higher position

$0.875 \times 2 = 1.75$ Integer part: 1

$0.75 \times 2 = 1.5$ Integer part: 1

$0.5 \times 2 = 1$ Integer part: 1

Lower position

Answer: $(0.875)_{10} = (0.111)_2$

Practice: $(0.6875)_{10} = (?)_2$

Converting decimal number into binary number

- For a decimal number that has both integer and fractional parts

- Convert the integer and fractional parts separately

- Example: $(215.6875)_{10} = (?)_2$

2. Fundamental Concepts

- **Programming Modes:**
  - **Interactive Mode:** Typing one line at a time in the Python Shell.
  - **Script Mode:** Writing code in a `.py` file and executing it.
- **Syntax:** Python has a strict set of rules. A `SyntaxError` means you violated them.
- **Program Flow:**
  - **Sequential:** Executing statements one after another.
  - **Conditional:** Skipping steps (e.g., `if`).
  - **Repeated (Loops):** Repeating code (e.g., `while`).
- **Comments:** Use `#` for notes or to temporarily disable code.

Interactive v.s. script

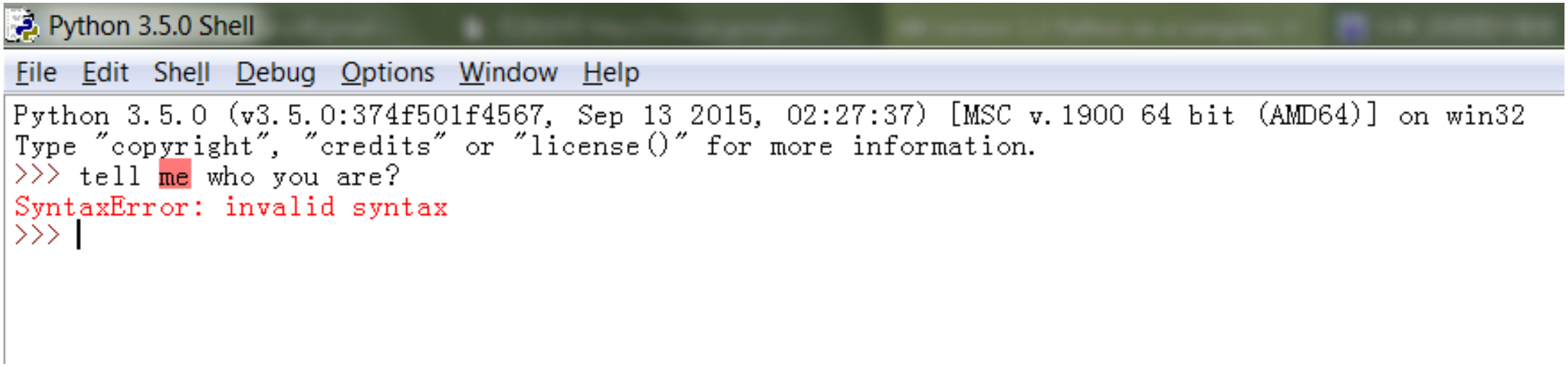
- Interactive

✓ You type directly to Python one line at a time and it responds

- Script

✓ You enter a sequence of statements (lines) into a file using a text editor and tell Python to execute the file

Syntax Error



Program steps or program flow

- Like a recipe, a program is a sequence of steps to be done in pre-determined order
- Some steps are conditional, i.e. they may be skipped
- Sometimes, we will repeat some steps
- Sometimes, we store a set of steps to be used over and over again in future as needed

Sequential flow

Execute sequentially

```
>>> x=2
>>> print(x)
2
>>> x=x*10
>>> print(x)
20
>>>
```

Outputs

- When a program is running, it flows from one step to the next
- We as programmers, set up “paths” for the program to follow

Conditional flow

Program

```
x=5
if x<10:
    print("smaller")
if x>20:
    print("bigger")
print("finished")
```

Outputs

Repeated flow

Program

```
n=5
while n>0:
    print(n)
    n = n - 1
print("Finish")
```

Outputs

- Loops (repeated steps) have iterative variables that change each time through a loop
- Often these iterative variables go through a sequence of numbers

3. Variables and Data Types

- **Variables:** A named space in memory to store data.
  - **Assignment:** `x = 10`
  - **Naming Rules:**
    - Must begin with a letter or `_`.
    - Can only contain letters, numbers, and `_`.
    - Case-sensitive (`apple` is not `Apple`).
- **Reserved Words:** You cannot use words like `if`, `while`, `False` as variable names.

- **Key Data Types:**
  - **Integers (`int`):** Whole numbers (`100` , `-50`).
  - **Floating-Point (`float`):** Numbers with decimals (`3.14` , `-0.5`).
  - **Strings (`str`):** Text in single `'` or double `"` quotes.
  - **Booleans (`bool`):** `True` or `False` .
- **Type Checking & Conversion:**
  - Check with `type()` .
  - Use `int()` , `float()` , `str()` to convert between types.
  - `input()` always returns a string!

Type conversion

- When an expression contains both integer and float, integers will be converted into float implicitly

- You can control this using functions int() and float()

- Example:

```
>>> print(float(99)/100)

>>> i=42
>>> type(i)

>>> f=float(i)
>>> print(f)

>>> type(f)

>>> print(1+2*float(3)/4-5)
```

String conversions

- You can also use int() and float() to convert strings into numbers

- You will get an error if the string contains characters other than numbers

Converting numbers into string

- We can convert numbers into string using function str()

```
>>> str(5)                    # Convert int to a string.
'5'
>>> str(1 + 10 + 100)         # Convert int expression to a string.
'111'
>>> str(-12.34)               # Convert float to a string.
'-12.34'
>>> str("Hello World!")       # str() accepts string arguments.
'Hello World!'
>>> str(divmod(14, 9))        # Convert tuple to a string.
'(1, 5)'
>>> x = 42
>>> str(x)                    # Convert int variable to a string.
'42'
```

4. Expressions and Operators

- **Expression:** A combination of values, variables, and operators that evaluates to a new value (e.g., `2 * (x + 3)`).
- **Arithmetic Operators:** `+`, `-`, `*`, `/`, `**`, `%`, `//`.
- **Comparison Operators:** `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **Logical Operators:** `and`, `or`, `not`.

- **Operator Precedence:**
    i. Parentheses `()`
    ii. Exponentiation `**`
    iii. Multiplication `*`, Division `/`, Remainder `%`
    iv. Addition `+`, Subtraction `-`
    v. Comparison Operators
    vi. Logical Operators
- **Assignment:** `x = 5`. Note: `==` is for comparison, `=` is for assignment.
- **Augmented Assignment:** `x += 7` is the same as `x = x + 7`.

Assignment statement

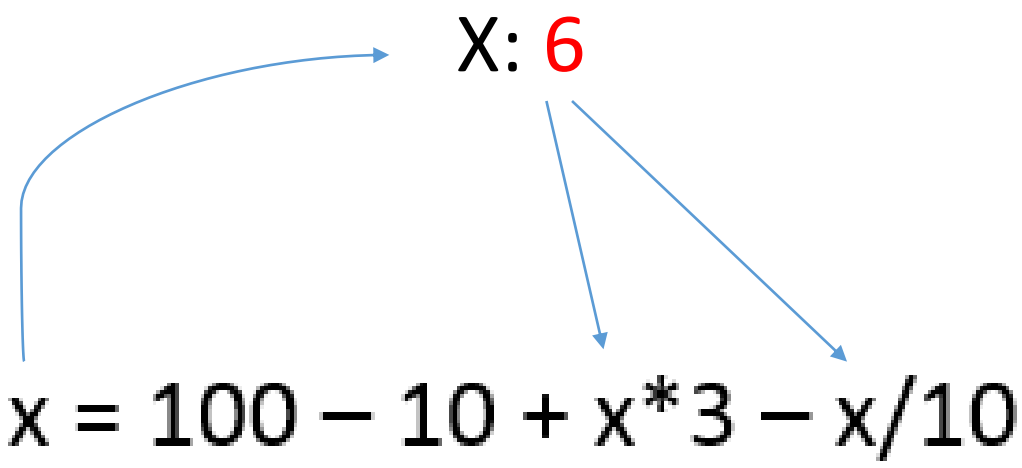
- We assign a value to a variable using the assignment operator (=)

- An assignment statement consists of an expression on the right hand side, and a variable to store the result

Example: x = 100 – 10 + x*3 – x/10

Assignment statement

- There is a location in the memory for x
- Whenever the value of x is needed, it can be retrieved from the memory
- After the expression is evaluated, the result will be put back into x



Assignment statement

- There is a location in the memory for x
- Whenever the value of x is needed, it can be retrieved from the memory
- After the expression is evaluated, the result will be put back into x

X: 107.4

x = 100 – 10 + x*3 – x/10

Cascaded assignment

- We can set multiple variables into the same value using a single assignment statement

Example

```
>>> z = y = x = 2 + 7 + 2
>>> x, y, z
(11, 11, 11)
```

Simultaneous assignment

- The values of two variables can be exchanged using simultaneous assignment

Example

```
>>> c = "deepSecret"          # Set current password.
>>> o = "you'll never guess" # Set old password.
>>> c, o                      # See what passwords are.
('deepSecret', "you'll never guess")
>>> c, o = o, c               # Exchange the passwords.
```

Bad use of simultaneous assignment

```
>>> # A bad use of simultaneous assignment.
>>> x, y = (45 + 34) / (21 - 4), 56 * 57 * 58 * 59
>>> x, y
(4.647058823529412, 10923024)
>>> # A better way to set the values of x and y.
>>> x = (45 + 34) / (21 - 4)
>>> y = 56 * 57 * 58 * 59
>>> x, y
(4.647058823529412, 10923024)
```

Operator precedence

Example: x = 1+2**3/4*5

Floor division

```
>>> time = 257              # Time in seconds.
>>> minutes = time // 60 # Number of complete minutes in time.
>>> print("There are", minutes, "complete minutes in", time, "seconds.")
There are 4 complete minutes in 257 seconds.
>>> 143 // 25
5
>>> 143.4 // 25
5.0
>>> 9 // 2.5
3.0
```

divmod()

```
>>> time = 257          # Initialize time.
>>> SEC_PER_MIN = 60 # Use a "named constant" for 60.
>>> divmod(time, SEC_PER_MIN)   # See what divmod() returns.
(4, 17)
>>> # Use simulataneous assignment to obtain minutes and seconds.
>>> minutes, seconds = divmod(time, SEC_PER_MIN)
>>> # Attempt to display the minutes and seconds in "standard" form.
>>> print(minutes, ":", seconds)
4 : 17
>>> # Successful attempt to display time "standard" form.
>>> print(minutes, ":", seconds, sep="")
4:17
>>> # Obtain number of quarters and leftover change in 143 pennies.
>>> quarters, cents = divmod(143, 25)
>>> quarters, cents
(5, 18)
```

5. Control Flow

- **Conditional Execution:**
  - `if`: Executes code only if a condition is `True`.
  - `if-else` **(Two-way):** Provides an alternative block if the condition is `False`.
  - `if-elif-else` **(Multi-way):** Tests multiple conditions in sequence.
  - **Nested Decisions:** An `if` statement inside another.
- **Indentation:** Defines code blocks. It is not just for style.
- **Truthiness:** `0`, `0.0`, and empty collections (`[]`, `""`) are treated as `False`. Most other values are `True`.

Conditional flow

Program

```
x=5
if x<10:
    print("smaller")
if x>20:
    print("bigger")
print("finished")
```

Outputs

```
smaller
finished
>>>
```

Comparison operators

- Boolean expressions ask a question and produce a Yes/No result, which we use to control program flow
- Boolean expressions use comparison operators to evaluate Yes/No or True/False
- Comparison operators check variables but do not change the values of variables
- Careful!! “=“ is used for assignment

x < y	Is x less than y?
x <= y	Is x less than or equal to y?
x == y	Is x equal to y?
x >= y	Is x greater than or equal to y?
x > y	Is x greater than y?
x != y	Is x not equal to y?

Comparison operators

```
x=5
if x==5:
    print("Equals 5")

if x>4:
    print("Greater than 4")

if x>=5:
    print("Greater than or equal to 5")

if x<=5:
    print("Less than or equal 5")

if x!=6:
    print("Not equal 6")
```

```
Equals 5
Greater than 4
Greater than or equal to 5
Less than or equal 5
Not equal 6
```

Examples of comparison

```
>>> 5 > 7                    # Is 5 greater than 7?
False
>>> x, y = 45, -3.0
>>> x > y                    # Is 45 greater than -3.0?
True
>>> result = x > y + 50 # Is 45 greater than -3.0 + 50?
>>> result
False
>>> if 1 + 1 > 1:
...     print("I think this should print.")
...
I think this should print.
```

Bool()

```
>>> x = 0; y = 0.0; z = 0 + 0j
>>> bool(x), bool(y), bool(z)
(False, False, False)
>>> x = -1; y = 1.e-10; z = 0 + 1j
>>> bool(x), bool(y), bool(z)
(True, True, True)
```

One way decisions

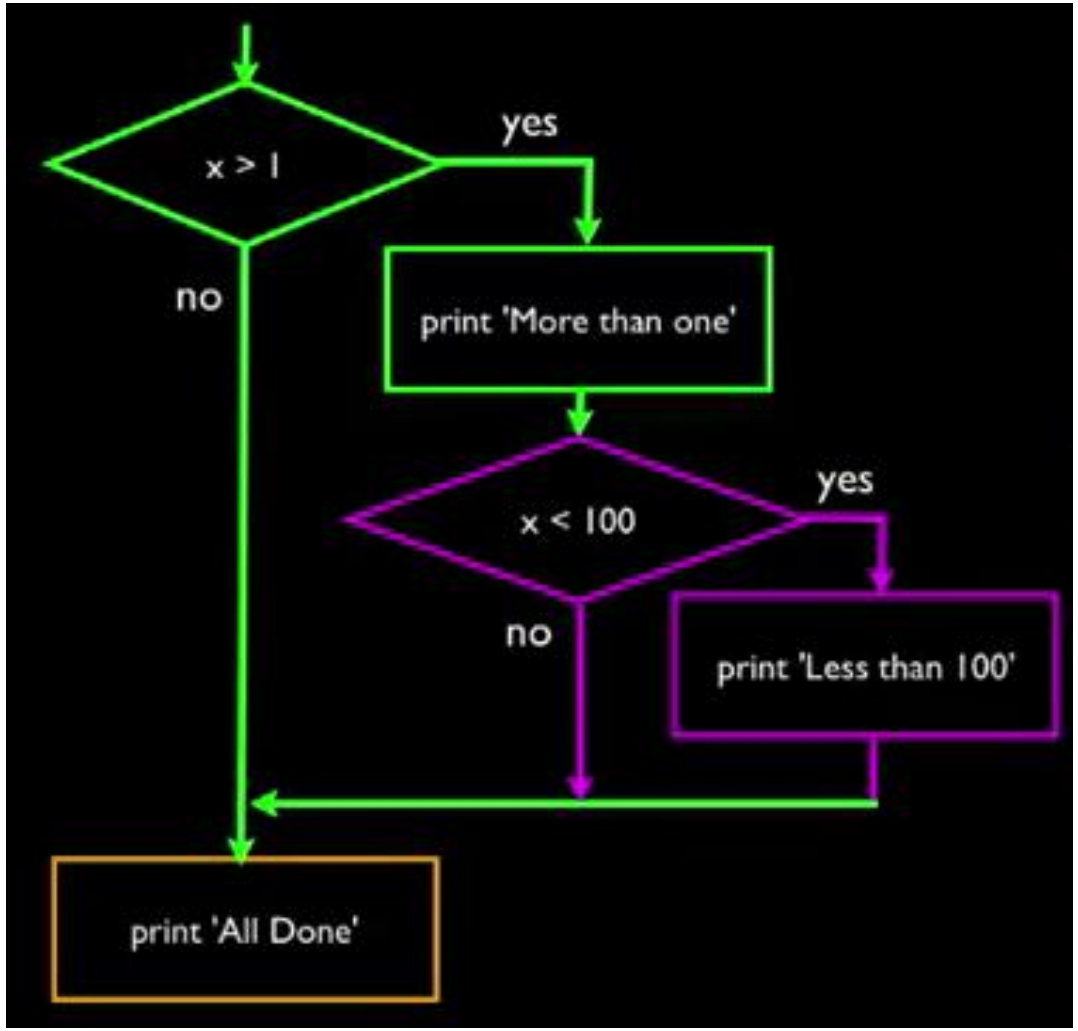
```
x=5
print('Before 5')
if x==5:
    print('Is 5')
    print('Is still 5')
    print('Third 5')

print('Afterwards 5')

print('Before 6')
if x==6:
    print('Is 6')
    print('Is still 6')
    print('Third 6')

print('Afterwards 6')
```

```
Before 5
Is 5
Is still 5
Third 5
Afterwards 5
Before 6
Afterwards 6
```

Nested decisions

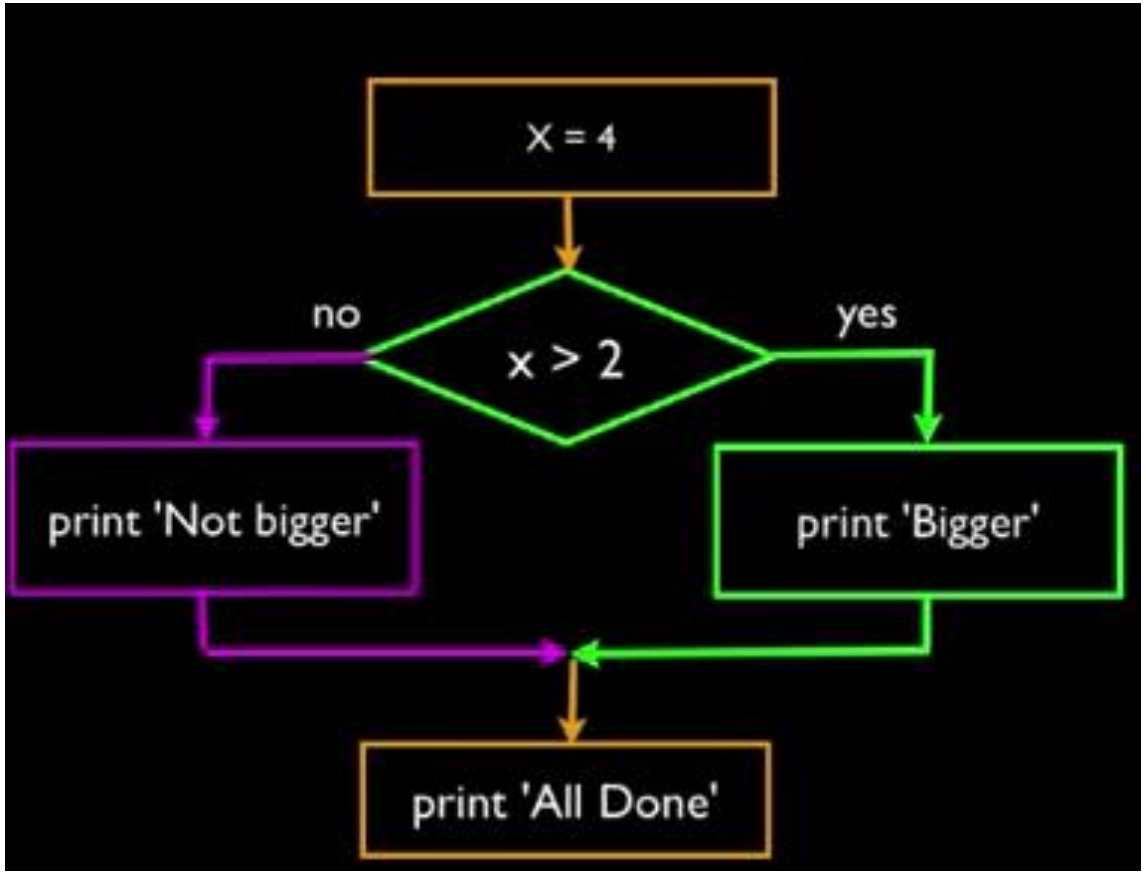
Example

```
x=42
if x>1:
    print('More than 1')

    if x<100:
        print('Less then 100')

print('Finished')
```



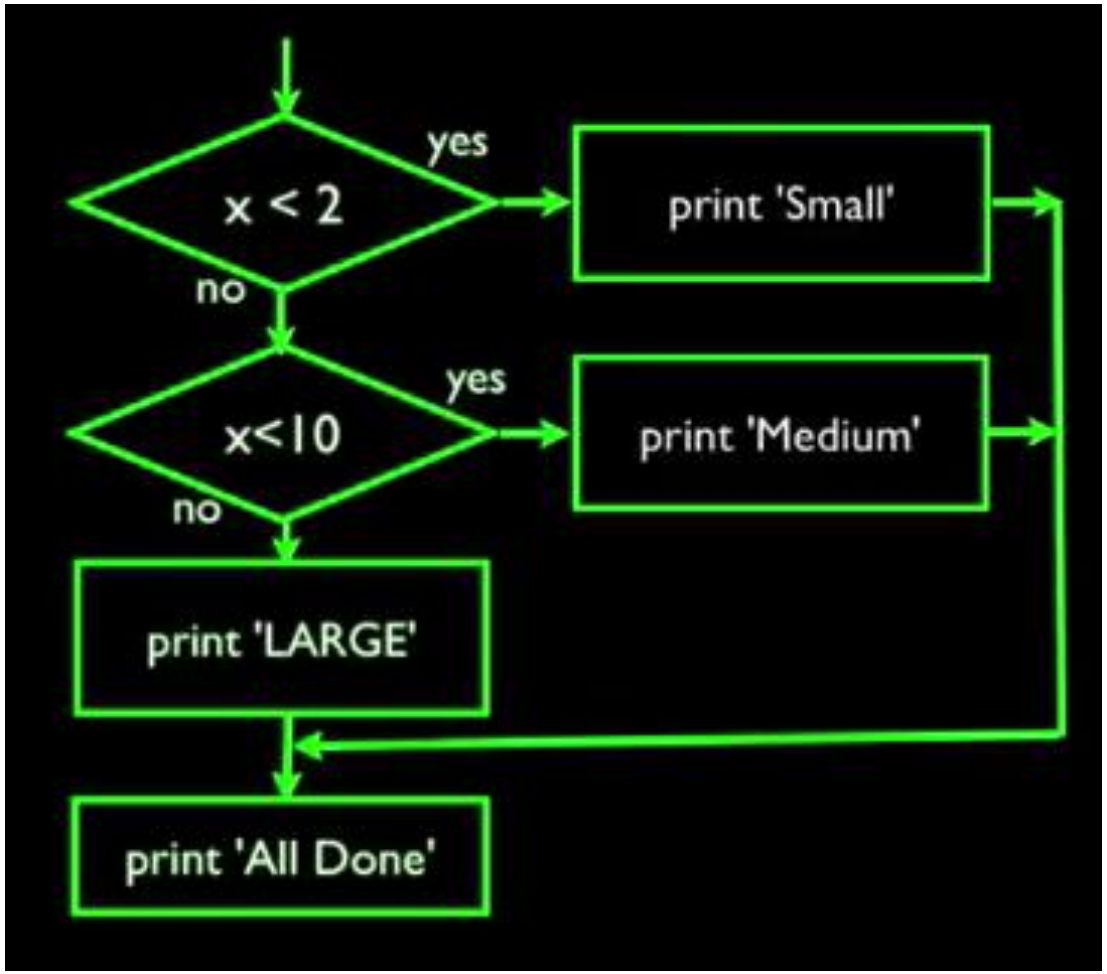
Two way decision using else

```
x=1

if x>2:
    print('Bigger')
else:
    print('Smaller')

print('Finished')
```



Multi-way decisions

```
x=2
if x<2:
    print('Small')
elif x<10:
    print('Medium')
else:
    print('Large')

print('Finished')
```



Which will never be printed?

```
x=4

if x<=2:
    print('Below 2')
elif x>2:
    print('Above 2')
else:
    print('Something else')
print('Finished')
```

```
x=8

if x<2:
    print('Below 2')
elif x<20:
    print('Below 20')
elif x<10:
    print('Below 10')
else:
    print('Something else')
print('Finished')
```

Example

```
>>> not True
False
>>> False and True
False
>>> not False and True
True
>>> (not False) and True      # Same as previous statement.
True
>>> True or False
True
```

Example

```
>>> not False or True          # Same as: (not False) or True.
True
>>> not (False or True)
False
>>> False and False or True   # Same as: (False and False) or True.
True
>>> False and (False or True)
False
```

6. Loops

- **`while` Loops (Indefinite):** Continues as long as a condition is `True`.
- **`for` Loops (Definite):** Iterates over each item in a sequence.
- **Loop Control Statements:**
  - **`break`:** Exits a loop immediately.
  - **`continue`:** Skips to the next iteration.
- **Common Loop Patterns:**
  - **Counting:** `count += 1`
  - **Summing (Accumulating):** `total += value`
  - **Finding Largest/Smallest:** Keep track of the "largest/smallest so far".
  - **Searching:** Use a boolean flag to check if a condition is met.

An infinite loop

```
n=5
while n>0:
    print('Lather')
    print('Rinse')
n=n-1
print('Dry off!')
```

- What is wrong with this program?

Definite loop

- Quite often we have a finite set of items
- We can use a loop, each iteration of which will be executed for each item in the set, using the for statement
- These loops are called “definite loops” because they execute an exact number of times
- It is said that “definite loops iterate through the members of a set”

Practice

- **Given a set of numbers, write a program to calculate their sum using for loop**

Answer

```
numberSet = [3, 4, 98, 38, 9, 10, 199, 78]

total = 0
print('Before', total)
for num in numberSet:
    total = total + num
    print(total, num)
print('Last', total)
```

```
Before 0
3 3
7 4
105 98
143 38
152 9
162 10
361 199
439 78
Last 439
```

Practice

- **Given a set of numbers, write a program to calculate their average using for loop**

Answer

```
numberSet = [3, 4, 98, 38, 9, 10, 199, 78]

total = 0
count = 0
print('Before', total)
for num in numberSet:
    total = total + num
    count = count + 1
    print(count, total, num)
print('Last', total, total/count)
```

```
Before 0
1 3 3
2 7 4
3 105 98
4 143 38
5 152 9
6 162 10
7 361 199
8 439 78
Last 439 54.875
```

Search using a Boolean variable

Example

```
found = False

print('Before', found)

for value in [9, 41, 12, 3, 74, 15]:
    if value == 74:
        found = True
    print(found, value)
print('After', found)
```

Output

```
Before False
False 9
False 41
False 12
False 3
True 74
True 15
After True
```

- If we want to search in a set and double check whether a specific number is in that set
- We can use a Boolean variable, set it to False at the beginning, and assign True to it as long as the target number is found

Finding the largest number

Example

```
largest_so_far = -1
print('Before',largest_so_far)

for num in [9,39,21,98,4,5,100,65]:
    if num>largest_so_far:
        largest_so_far = num
    print(largest_so_far,num)

print('After',largest_so_far)
```

Output

```
Before -1
9 9
39 39
39 21
98 98
98 4
98 5
100 100
100 65
After 100
```

- Use a variable to store the largest number we have seen so far
- If the current number is larger, we assign it to the store variable

Finding the smallest number

```
smallest_so_far = -1
print('Before',smallest_so_far)

for num in [9,39,21,98,4,5,100,65]:
    if num < smallest_so_far:
        smallest_so_far = num
    print(smallest_so_far,num)

print('After',smallest_so_far)
```

- Use a variable to store the smallest number we have seen so far
- If the current number is smaller, we assign it to the store variable
- What is the problem with this program?

Finding the smallest number

Example

```
smallest_so_far = None
print('Before',smallest_so_far)

for num in [9,39,21,98,4,5,100,65]:
    if smallest_so_far == None:
        smallest_so_far = num
    elif num < smallest_so_far:
        smallest_so_far = num
    print(smallest_so_far,num)

print('After',smallest_so_far)
```

Output

```
Before None
9 9
9 39
9 21
9 98
4 4
4 5
4 100
4 65
After 4
```

- We still use a variable to store the smallest value seen so far
- In the first iteration, the smallest value is none, so we need to use an if statement to check this

The is and is not operator

```
smallest_so_far = None
print('Before',smallest_so_far)

for num in [9,39,21,98,4,5,100,65]:
    if smallest_so_far is None:
        smallest_so_far = num
    elif num < smallest_so_far:
        smallest_so_far = num
    print(smallest_so_far,num)

print('After',smallest_so_far)
```

- Python has a "is" operator which can be used in logical expression
- Implies "is the same as"
- Similar to, but stronger than ==
- "is not" is also an operator

More about Bool()

```
>>> x = 0; y = 0.0; z = 0 + 0j
>>> bool(x), bool(y), bool(z)
(False, False, False)
>>> x = -1; y = 1.e-10; z = 0 + 1j
>>> bool(x), bool(y), bool(z)
(True, True, True)
>>> x = []; y = [0]; z = "0"
>>> bool(x), bool(y), bool(z)
(False, True, True)
```

7. Functions

- **Defining and Calling:**
  - Use `def function_name(parameter1, parameter2):`.
  - Call with `function_name(arg1, arg2)`.
- **Return Values:**
  - Use `return` to send a result back.
  - **Fruitful functions** return a value; **void functions** do not (they return `None`).
- **Arguments:**
  - **Default Arguments:** `def greet(name, message="Hello"):`
- **Variable Scope:**
  - **Local:** Defined inside a function.
  - **Global:** Defined outside any function. Use the `global` keyword to modify one from inside a function.

A sample code

Program

```
x=5
print('Hello')


def print_lyrics():
    print('I am a lumberjack, and I am okay.')
    print('I sleep all night and I work all day.')

print('Yo')
print_lyrics()
x=x+2
print(x)
```

Output

```
Hello
Yo
I am a lumberjack, and I am okay.
I sleep all night and I work all day.
7
```

Argument, parameter, and result

```
>>> big = max('I am the one')
>>> print(big)
t
```

Parameter

Result

```
'I am the one'
```

```
def max(inp):
    blah
    blah
    for x in y:
        blah
        blah
    return 't'
```

```
t
```

Argument

Functions without return

- When a function has no return statement, it will return None

```
# Print grade for the score
def printGrade(score):
    if score >= 90.0:
        print('A')
    elif score >= 80.0:
        print('B')
    elif score >= 70.0:
        print('C')
    elif score >= 60.0:
        print('D')
    else:
        print('F')

def main():
    score = eval(input("Enter a score: "))
    print("The grade is ", end = " ")
    printGrade(score)

main() # Call the main function
```

Scope of variables

- The scope of a variable is the part of program where this variable can be accessed
- A variable created inside a function is referred to as a local variable
- Global variables are created outside all functions and are accessible to all functions in their scope

```
globalVar = 1
def f1():
    localVar = 2
    print(globalVar)
    print(localVar)

f1()
print(globalVar)
print(localVar) # Out of scope, so this gives an error
```

Scope of variables

- Different variables may share a name if they have different scopes

```
x = 1
def f1():
    x = 2
    print(x) # Displays 2

f1()
print(x) # Displays 1
```

Global variable

- In a function, you can use keyword global to specify that a variable is a global variable

- Be very careful when define and use global variable

```
x = 1
def increase():
    global x
    x =  x + 1
    print(x) # Displays 2

increase()
print(x) # Displays 2
```

```
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Default argument

- Python allows you to define functions with default argument values
- The default argument values will be passed to the function, when it is invoked without arguments

```
def printArea(width = 1, height = 2):
    area = width * height
    print("width:", width, "\theight:", height, "\tarea:", area)

printArea() # Default arguments width = 1 and height = 2
printArea(4, 2.5) # Positional arguments width = 4 and height = 2.5
printArea(height = 5, width = 3) # Keyword arguments width
printArea(width = 1.2) # Default height = 2
printArea(height = 6.2) # Default width = 1
```

Return multiple values

- Python allows a function to return multiple values

- The sort function returns two values; when it is invoked, you need to pass the returned values in a simultaneous assignment

```
def sort(number1, number2):
    if number1 < number2:
        return number1, number2
    else:
        return number2, number1

n1, n2 = sort(3, 2)
print("n1 is", n1)
print("n2 is", n2)
```

8. Built-in Functions & Error Handling

- **Common Functions:**
  - `print()` : Outputs values to the console.
  - `input()` : Prompts the user for input (returns a string).
  - `len()` : Returns the length/size of an object.
  - `eval()` : Executes a string as a Python expression.
  - `divmod()` : Returns quotient and remainder as a pair.
- **Error Handling:**
  - **`try` and `except`** : Handle code that might fail without crashing.

```
try:
    value = int("abc")
except ValueError:
    print("That was not a number!")
```

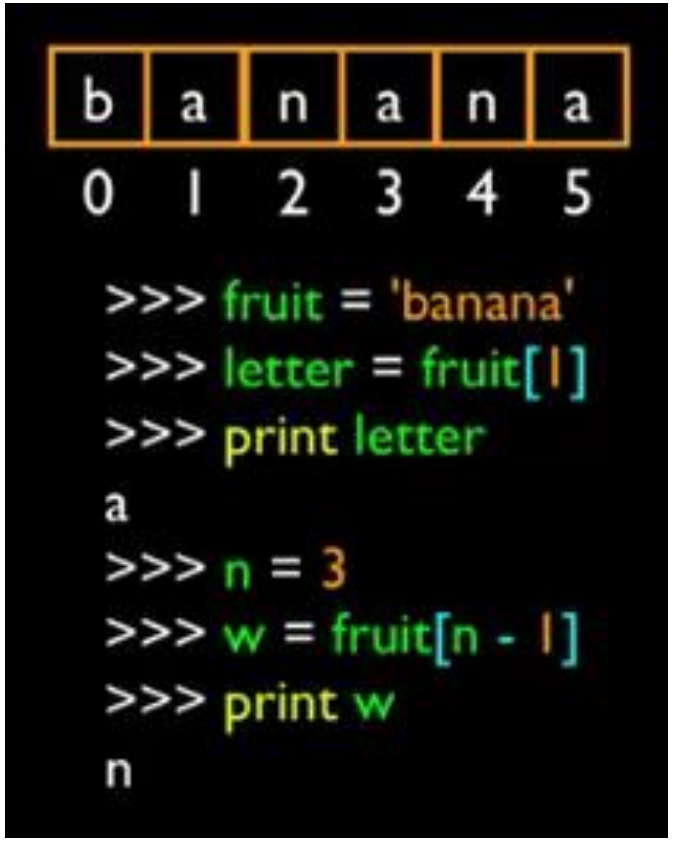
9. Strings

- Strings are **immutable** sequences of characters.
- **Indexing & Slicing:** `my_string[0]`, `my_string[start:end]`.
- **Operators:**
  - `+` (Concatenation): `'hello' + 'world'`
  - `*` (Repetition): `'Go' * 3`
  - `in` (Membership): `'sub' in 'substring'`

- **Common Methods:**
  - `.lower()` / `.upper()`: Change case.
  - `.strip()` / `.lstrip()` / `.rstrip()`: Remove whitespace.
  - `.startswith(prefix)` / `.endswith(suffix)`.
  - `.find(substring)`: Returns index or -1.
  - `.replace(old, new)`: Returns a new string with replacements.
  - `.split(delimiter)`: Breaks a string into a list.

Looking inside strings

- We can get any character in a string using an index specified in square brackets
- The index value must be an integer which starts from zero
- The index value can be an expression



Index out of range

- You will get an Python error if you attempt to index beyond the end of a string

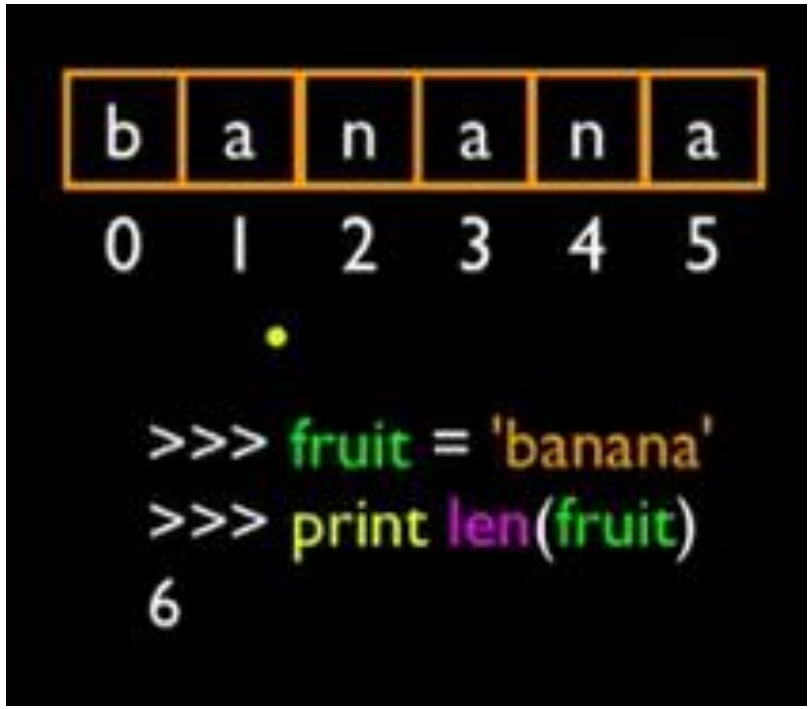
```
a = 'hi'
print(a[2])
```

```
Traceback (most recent call last):
  File "/Users/pinjia/Documents/aie1001/codes/week4.py", line 221, in <
  module>
    print(a[2])
IndexError: string index out of range
[Finished in 41ms with exit code 1]
```

- Be careful when specifying an index value

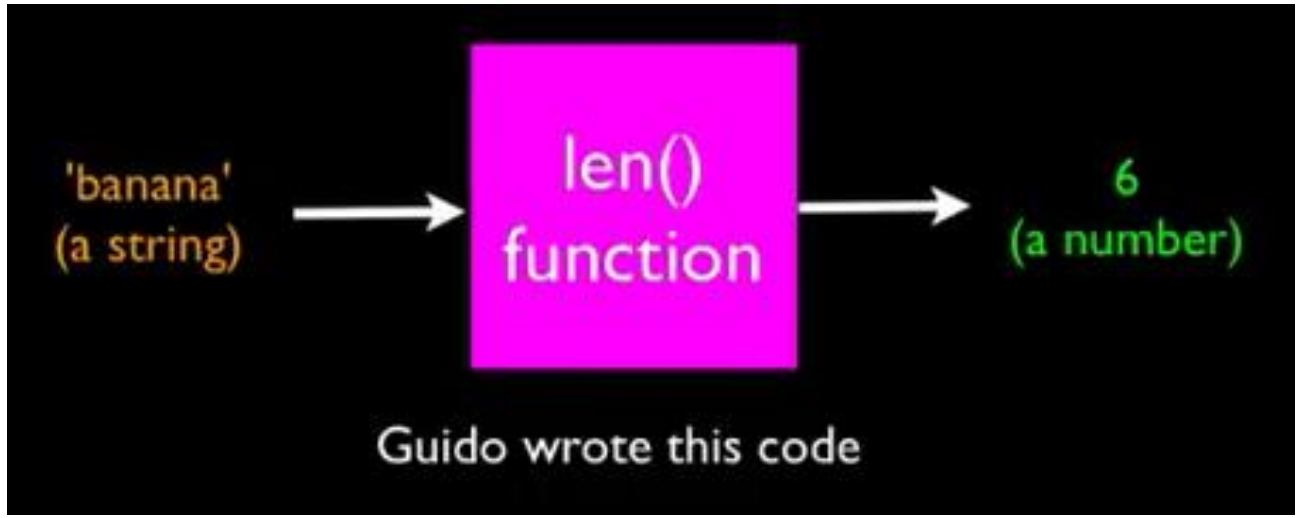
Strings have length

- There is a built-in function len() which gives us the length of a string



Len() function

```
>>> fruit = 'apple'
>>> length = len(fruit)
>>> print(length)
5
```



Looping through strings

- Using a for statement, we can easily loop through each character in a string

- String is essentially a list in Python

```
fruit = 'I am the one,Morpheus'

n = 0
for i in fruit:
    print(n, i)
    n=n+1

print('finished')
```

```
0 I
1
2 a
3 m
4
5 t
6 h
7 e
8
9 o
10 n
11 e
12 ,
13 M
14 o
15 r
16 p
17 h
18 e
19 u
20 s
finished
```

Practice

- Write a program to use a while statement together with len() function to loop through a given string

Example

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2016'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
```

10. Collections: Lists, Tuples, Dictionaries

- **`is` vs. `==`:**
  - `==` checks if values are equal.
  - `is` checks if two variables refer to the exact same object in memory.
- **Simultaneous Assignment:** `x, y = 10, 20`. Can be used to swap variables: `x, y = y, x`.

Lists `[]`

- An ordered, **mutable** (changeable) collection.
- **Creation:** `my_list = [1, 'apple', 2.5]`
- **Access:** `my_list[0]`, `my_list[1:3]` (slicing).
- **Common Methods:**
  - `.append(item)`: Add to the end.
  - `.sort()`: Sort the list in-place.
  - `.reverse()`: Reverse the list in-place.

How long is a list?

- The len() function takes a list as input and returns the number of elements in that list

- Actually len() tells us the number of elements in any sequence (e.g. strings)

```
>>> greet = 'Hello Bob'
>>> print(len(greet))
9
>>> x=[1, 2, 'joe', 99]
>>> print(len(x))
4
```

Range() function

- The range() function returns a list of numbers

- We can construct an index loop using for and an integer iterator

```
>>> x=range(4)
>>> x
range(0, 4)
>>> x[0]
0
>>> x[1]
1
>>> x[2]
2
>>> x[3]
3
>>> x=range(2, 10, 2)
>>> x[0]
2
>>> x[3]
8
>>> x[4]
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    x[4]
IndexError: range object index out of range
```

Range() function

- What is the output of the following code?
- squares = [x * x for x in range(5) if x % 2 == 0]
  - A. [0, 1, 4, 9, 16]
  - B. [0, 2, 4]
  - C. [0, 4, 16]
  - D. [0, 4, 8]

A tale of two loops

Example

```
friends = ['Tom','Jerry','Bat']

for friend in friends:
    print('Happy new year,', friend)

for i in range(len(friends)):
    friend = friends[i]
    print('Happy new year,', friend)
```

Output

```
Happy new year, Tom
Happy new year, Jerry
Happy new year, Bat
Happy new year, Tom
Happy new year, Jerry
Happy new year, Bat
>>>
```

Concatenating lists using +

- Similar to strings, we can add two existing lists together to create a new list

```
>>> a=[1, 2, 3]
>>> b=[4, 5, 6]
>>> c=a+b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3]
```

Lists can be sliced using :

- Remember: similar to strings, the second number is "up to but no including"

```
>>> t=[9, 41, 12, 3, 74, 15]
>>> t[1:3]
[41, 12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

Built-in functions and lists

- There are a number of functions built into Python that take lists as inputs

- Remember the loops we built? These are much simpler

```
>>> numbers = [3, 41, 12, 9, 74, 15]
>>> print(len(numbers))
6
>>> print(max(numbers))
74
>>> print(min(numbers))
3
>>> print(sum(numbers))
154
>>> print(sum(numbers)/len(numbers))
25.666666666666668
```

Averaging with a list

```
total = 0
count = 0
while True:
    inp = input('Enter a number:')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total/count
print('The average is:', average)
```

Practice

- Write a program to instruct the user to input several numbers and calculate their average using list methods

Practice

- The header of an email takes the following format:

From professor.xman@uct.edu Sat Jan 5 09:14:16 2008

For a given email header, write a program to find out the domain of email address, and the month in which this email is sent

The double split pattern

- Sometimes we split a line one way, and then grab one piece of the line and split it again

From professor.xman@uct.edu Sat Jan 5 09:14:16 2008

```
words = header.split()
address = words[1].split('@')
```

Tuples `( )`

- An ordered, **immutable** (not changeable) sequence.
- **Efficient:** More memory and performance efficient than lists.
- **Use Case:** Good for temporary data or data that should not change, like keys in a dictionary or returning multiple values from a function.
- Cannot sort, append, or reverse a tuple after creation.

Tuples are more efficient

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists

- In our program when we are making “temporary variables” we prefer tuples over lists

Tuples and dictionaries

- The item() method in dictionaries returns a list of (key, value) tuples

```
>>> d=dict()
>>> d['csev']=2
>>> d['cwen']=4
>>> for (k,v) in d.items():
        print(k,v)


csev 2
cwen 4
>>> tups = d.items()
>>> print(tups)
dict_items([('csev', 2), ('cwen', 4)])
>>> print(list(tups))
[('csev', 2), ('cwen', 4)]

>>> tups = list(tups)
>>> tups[1]
('cwen', 4)
```

Tuples are comparable

- The comparison operators work with tuples and other sequences if the first item is equal. Python goes on to the next element, until it finds the elements which are different

```
>>> (0, 1, 2)<(5, 1, 2)
True
>>> (0, 1, 200000)<(0, 3, 4)
True
>>> ('Jones','Sally')<('Jones','Fred')
False
>>> ('Jones','Sally')>('Adams','Sam')
True
```

Using sorted()

- We can also do this using a built-in function sorted() which takes a sequence as a parameter and returns a sorted sequence

```
d = {'c':10, 'b':1, 'a':22}
t = d.items()

w = sorted(t)
print(w)
print(t)
```

```
[('a', 22), ('b', 1), ('c', 10)]
dict_items([('c', 10), ('b', 1), ('a', 22)])
```

Practice

- Write a program, which sorts the elements of a dictionary by the value of each element

Sort by values instead of key

- If we could construct a list of tuples of the form (key, value) we could sort by value

- We do this with a for loop that creates a list of tuples

```
>>> d={'a':10,'b':1,'c':22}
>>> tmp = list()
>>> for k,v in d.items():
        tmp.append((v,k))
```

```
>>> print(tmp)
[(22, 'c'), (1, 'b'), (10, 'a')]
>>> tmp.sort(reverse=True)
>>> print(tmp)
[(22, 'c'), (10, 'a'), (1, 'b')]
```

Example: Finding the 10 most common words in a file

```
fhand = open('myhost.txt','r')
counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0)+1

lst = list()
for key,val in counts.items():
    lst.append((val,key))

lst.sort(reverse = True)

for val,key in lst[:10]:
    print(key,val)
```

Dictionaries `{}`

- A collection of **key-value** pairs.
- Values are accessed via their **key**, not an index.
- **Working with Dictionaries:**
  - Add/Update: `my_dict['key'] = value`
  - Access: `my_dict['key']` (causes `KeyError` if key doesn't exist).
  - Safe Access: `my_dict.get(key, default_value)`.
- **Methods & Looping:**
  - `.keys()`, `.values()`, `.items()`.
  - `for key in my_dict:`
  - `for key, value in my_dict.items():`

List v.s. dictionary

- Dictionaries are similar to lists, except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age']=21
>>> ddd['course']=182
>>> print(ddd)
{'age': 21, 'course': 182}
>>> ddd['age']=23
>>> print(ddd)
{'age': 23, 'course': 182}
```

Counting with a dictionary

- A common use of dictionary is counting how often we “see” something

```
>>> ccc=dict()
>>> ccc['csev']=1
>>> ccc['cwen']=1
>>> print(ccc)
{'csev': 1, 'cwen': 1}
>>> ccc['cwen']=ccc['cwen']+1
>>> print(ccc['cwen'])
2
```

Dictionary tracebacks

- It is an error to reference a key which is not in the dictionary
- We can use the in operator to see if a key is in the dictionary

```
>>> ccc=dict()
>>> print(ccc['csev'])
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    print(ccc['csev'])
KeyError: 'csev'
>>> 'csev' in ccc
False
```

Practice

- Write a program to instruct the user to continuously input some words, and use dictionary to count how many times a word has been inputted before.

The get() method

- This pattern of checking to see if a key is already in a dictionary, and assuming a default value if the key is not there is so common, that there is a method called get() that does this for us

```
>>> counts = {'aaa':1,'bbb':2,'ccc':5}
>>> print(counts.get('eee',0))
0
```

Practice

- Write a program to instruct the user to input a line of texts, and use dictionary to count how many times a word has been seen in this line. You should use the get() method in this program.

Definite loops and dictionaries

- Even though dictionaries are not stored in order, we can write a for loop that goes through all elements in a dictionary – actually it goes through all the keys in that dictionary and looks up the values

```
counts = {'chuck':1,'fred':42,'jan':100}

for key in counts:
    print(key,counts[key])
```

```
jan 100
fred 42
chuck 1
```

Retrieving lists of keys and values

- You can get a list of keys, values or items (both) from a dictionary

```
>>> jjj = {'chuck':1,'fred':42,'jan':100}
>>> print(list(jjj))
['jan', 'fred', 'chuck']

>>> print(list(jjj.keys()))
['jan', 'fred', 'chuck']

>>> print(list(jjj.values()))
[100, 42, 1]
>>> print(list(jjj.items()))
[('jan', 100), ('fred', 42), ('chuck', 1)]
```

Bonus: two iteration variables

- We loop through the key-value pairs in a dictionary using two iteration variables

- Each iteration, the first variable is the key, and the second variable is the corresponding value for the key

```
counts = {'chuck':1,'fred':42,'jan':100}
for key,value in counts.items():
    print(key,value)
```

```
chuck 1
fred 42
jan 100
```

Question:

Which statement will cause an error?

- A. my_list = [1,2]; my_list.append(1)
- B. my_list = [1,2]; my_list[0] = 2
- C. my_tuple = (1,2); my_tuple[0] = 3
- D. my_dict = {'a':1}; my_dict['b'] = 2

Question:

Which of the following code **segments** will execute **without** raising an error?

A. d = {"name": "Alice"}; print(d["age"])

B. s = [1, 2]; s.append(2)

C. result = "hello" + ["world"]

D. my_list = [1]; my_list.append([1])

E. val = int("abc")

11. File Handling

- **Opening a File:**
  - `file_handle = open(filename, mode)`
  - **Modes:** `'r'` for reading (default), `'w'` for writing
- **Reading from a File:**
  - Loop line-by-line: `for line in file_handle:`
  - Read entire file: `contents = file_handle.read()`
- **Writing to a File:**
  - `file_handle.write("Some text\n")`
- **Closing a File:**
  - **Always** close the file to release resources: `file_handle.close()`
  - Better: Use `with open(...) as f:`, which closes the file automatically.

Practice

- Write a program to open a file and count how many lines are included in this file

Reading the whole file

- We can read the whole file into a single string

```
fhand = open('myhost.txt','r')
allText = fhand.read()
print('The length of the file:',len(allText))
print('The first 20 characters of the file:', allText[:20])
```

Searching through a file

- We can put an if statement in the for loop to print the lines which satisfy certain conditions

```
fhand = open('myhost.txt','r')

for line in fhand:
    if line.startswith('#')==True:
        print(line)

print('finished.')
fhand.close()
```

Writing to a file

- To write a file, use the open() function with ‘w’ argument

- Use the write() method to write to the file

```
fhand = open('test.txt','w')
fhand.write('The first line\n')
fhand.write('The second line\n')
fhand.write('The third line\n')
fhand.close()
```

12. Recursion

- **Definition:** A technique where a function calls itself to solve a problem.
- **Structure of a Recursive Function:**
  - **Base Case:** A condition that **stops** the recursion. Without it, you get infinite recursion.
  - **Recursive Step:** Breaks the problem down and calls the function on a smaller version.
- **Types of Recursion:**
  - **Linear Recursion:** Makes at most **one** new recursive call (e.g., Factorial, Binary Search).
  - **Multiple Recursion:** May make **two or more** recursive calls (e.g., Fibonacci).

Example: The factorial function

- The factorial of a positive integer n, denoted n!, is defined as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

The recursive definition of factorial function

- The factorial function can be naturally defined in a recursive way, for example, $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot 4!$

- More generally, for a positive integer n, we can define n! to be $n \cdot (n-1)!$

- Therefore, the recursive definition of factorial function is:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

Solution

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

What does the `factorial(4)` call return?

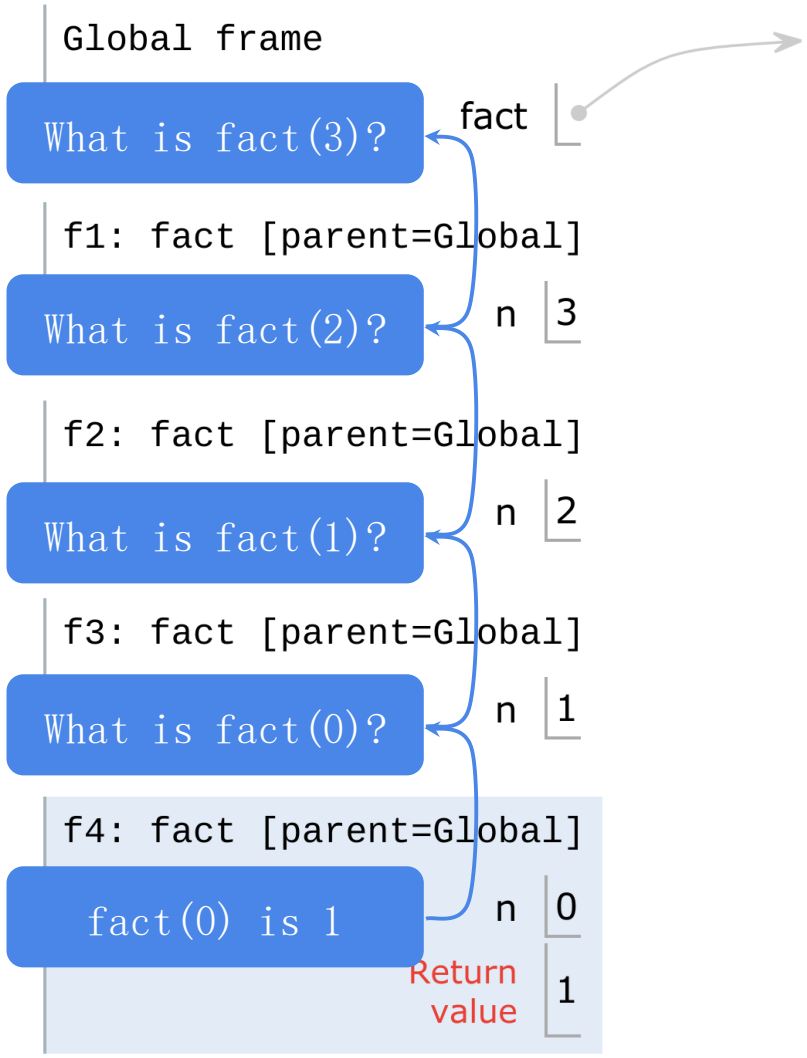
Recursion in environment diagram

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n - 1)
6
7  fact(3)
```

The same function fact is called multiple times, each time solving a simpler problem

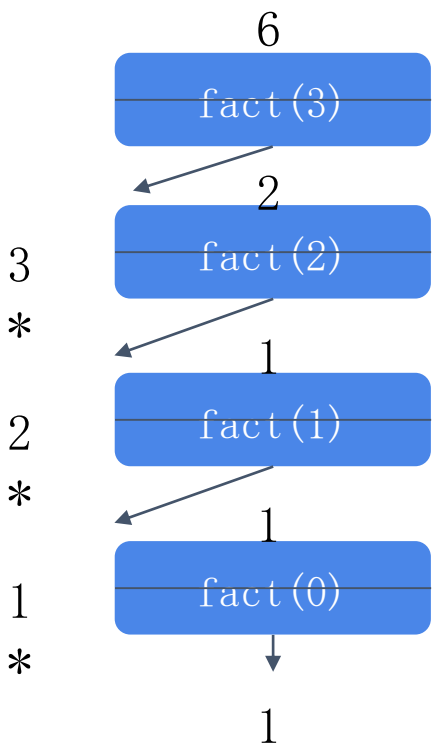
All the frames share the same parent - only difference is the argument

What n evaluates to depends upon the **current environment**



Recursive tree - another way to visualize recursion

```python
def fact(n):
    """Calculates n!"""
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```



Example: Binary Search

- **Purpose:** Efficiently search a **SORTED** sequence.
- **Strategy (Divide and Conquer):**
    i. Compare the target to the middle element.
    ii. If it's a match, you're done.
    iii. If the target is smaller, recur on the left half.
    iv. If the target is larger, recur on the right half.
- **Base Case:** The sequence is empty (target not found).

The strategy of binary search

- We call an element of the sequence a candidate if, at the current stage of the search, we cannot rule out that this item matches the target

- The algorithm maintains two parameters, low and high, such that all the candidate entries have index at least low and at most high

- Initially, low = 0 and high = n−1. We then compare the target value to the median candidate, that is, the item data[mid] with index

$$mid = \lfloor (low+high)/2 \rfloor$$

The strategy of binary search

- If the target equals data[mid], then we have found the item we are looking for, and the search terminates successfully

- If target < data[mid], then we recur on the first half of the sequence, that is, on the interval of indices from low to mid−1

- If target > data[mid], then we recur on the second half of the sequence, that is, on the interval of indices from mid+1 to high

Solution

```
def binarySearch(data, target, low, high):
    if low>high:
        print('Cannot find the target number!')
        return False
    else:
        mid = (low+high)//2
        if target==data[mid]:
            print('The target number is at position',mid)
            return True
        elif target<data[mid]:
            return binarySearch(data, target, low, mid-1)
        else:
            return binarySearch(data, target, mid+1, high)

def main():
    data = [1, 3, 5, 6, 16, 78, 100, 135, 900]
    target = 16
    binarySearch(data, target, 0, len(data)-1)
```

Practice: Power function

- Write a program to calculate the power function $f(x, n) = x^n$ using Recursion.

x*x*x* … *x ?

A better recursive definition of power function

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

Solution:

```
def myPower(x,n):
    if n==0:
        return 1
    else:
        partial = myPower(x,n//2)
        result = partial * partial
        if n%2==1:
            result = result * x
        return result
```

Example: Drawing an English ruler

- We denote the length of the tick designating a whole inch as the ***major tick length***.

- Between the marks for whole inches, the ruler contains a series of ***minor ticks***, placed at intervals of 1/2 inch, 1/4 inch, and so on.

- As the size of the interval decreases by half, the tick length decreases by one

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

(a)

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

(b)

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

(c)

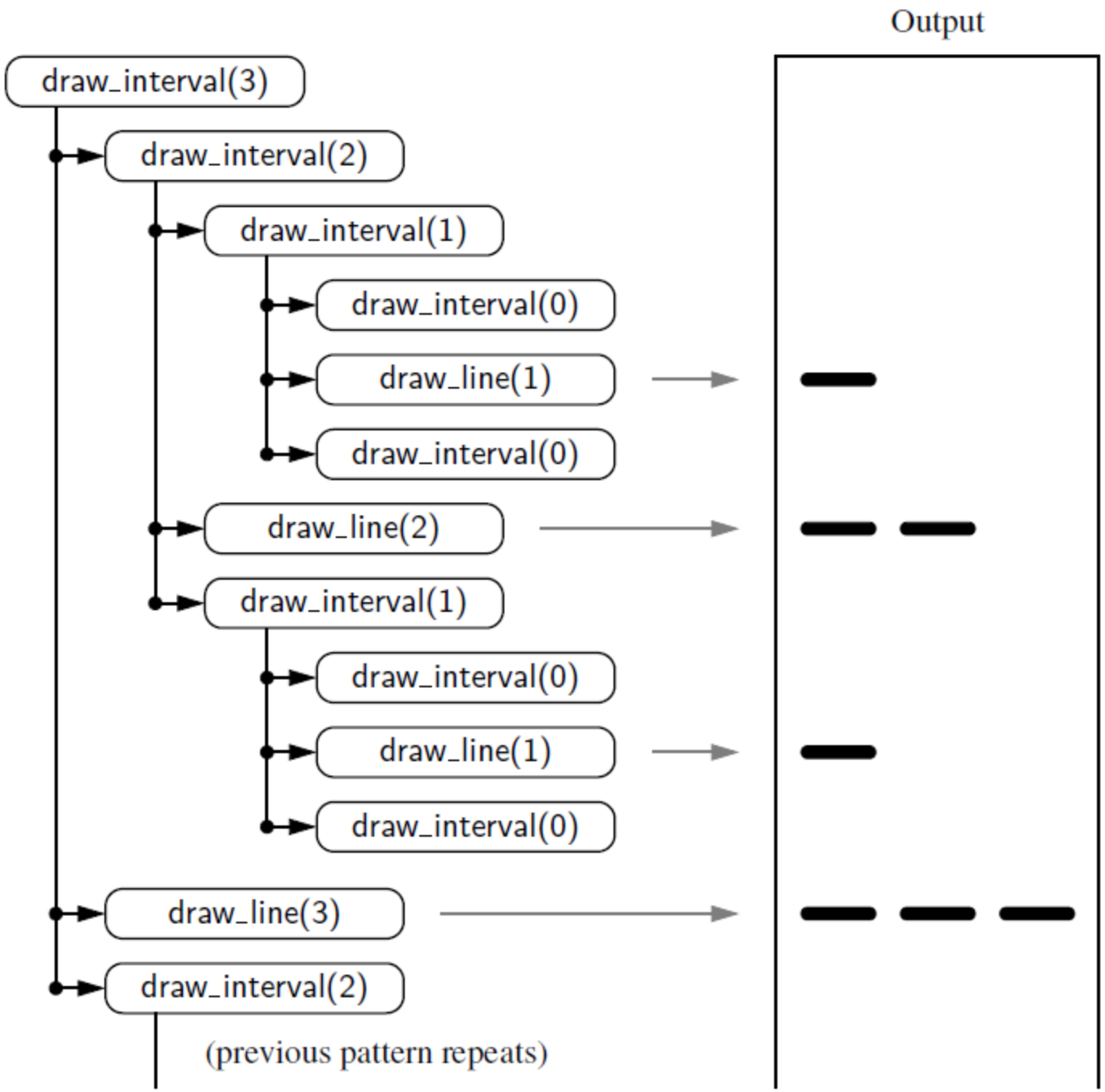
Recursive implementation of English ruler

- An interval with a central tick length $L \geq 1$ is composed of:
  - ✓ An interval with a central tick length $L-1$
  - ✓ A single tick of length L
  - ✓ An interval with a central tick length $L-1$

Solution

```
def draw_line(tickLen, tickLabel=''):
    line = '-'*tickLen
    if tickLabel:
        line+=' '+tickLabel
    print(line)

def draw_interval(centerLen):
    if centerLen>0:
        draw_interval(centerLen-1)
        draw_line(centerLen)
        draw_interval(centerLen-1)

def draw_ruler(numInch, majorLen):
    draw_line(majorLen, '0')

    for j in range(1, 1+numInch):
        draw_interval(majorLen-1)
        draw_line(majorLen, str(j))
```

The recursive trace for English ruler



Linear Recursion

- If a recursive function is designed so that each invocation of the body makes at most one new recursive call, this is known as linear recursion

- Finding the smallest number and binary search are both linear recursive algorithms

Linear Recursion

```
def binarySearch(data, target, low, high):
    if low > high:
        print ('Cannot find the target number!')
    else:
        mid = (low+high) // 2
        if target==data[mid]:
            print('The target number is at position', mid)
            return True
        elif target<data[mid]:
            # print ('We are searching position:', low, 'to', mid-1)
            return binarySearch(data, target, low, mid-1)
        else:
            # print ('We are searching position:', mid+1, 'to', high)
            return binarySearch(data, target, mid+1, high)
```

Practice: Sum of a list

- Given a list of numbers, write a program to calculate the sum of this list using recursion

Solution:

```
def linearSum(L, n):
    if n==0:
        return 0
    else:
        return linearSum(L, n-1)+L[n-1]

def main():
    L = [1, 2, 3, 4, 5, 9, 100, 46, 7]
    print('The sum is:', linearSum(L, len(L)))
```

Multiple recursion

- When a function makes two or more recursive calls, we say that it uses multiple recursion

- Drawing the English ruler is a multiple recursion program

Multiple recursion

- When a functio ve calls, we say that it us
- Drawing the Eng sion program

```
def draw_line(tickLen, tickLabel=''):
    line = '-'*tickLen
    if tickLabel:
        line+=' '+tickLabel
    print(line)

def draw_interval(centerLen):
    if centerLen>0:
        draw_interval(centerLen-1)
        draw_line(centerLen)
        draw_interval(centerLen-1)

def draw_ruler(numInch, majorLen):
    draw_line(majorLen, '0')

    for j in range(1, 1+numInch):
        draw_interval(majorLen-1)
        draw_line(majorLen, str(j))
```

Practice: Binary sum

- Write a function binarySum() to calculate the sum of a list of numbers. Inside binarySum() two recursive calls should be made

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Practice: Binary sum

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

```
binarySum(L,start,mid)+binarySum(L,mid,stop)
```

Practice: Binary sum

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

```
elif start==stop-1:
    return L[start]
```

Solution:

```
def binarySum(L, start, stop):
    if start>=stop:
        return 0
    elif start==stop - 1:
        return L[start]
    else:
        mid = (start+stop)//2
        return binarySum(L,start,mid)+binarySum(L,mid,stop)

def main():
    L = [1,2,3,4,5,6,7]
    print(binarySum(L,0,len(L)))
```

Example: Fault localization

There are **four** errors in the following code, which is intended to find the most frequent number (mode) in a list. Identify the lines or concepts that are incorrect.

```
1  def find_mode(data_list):
2      counts = {}
3      for item in data_list:
4          counts[item] = counts.get(item) + 1
5
6      max_count = 0
7      mode = None
8      for key, value in counts.items:
9          if value > max_count
10             max_count = value
11             mode = key
12
13     print("The mode is: " + mode)
14     return mode
```

Example: Code Completion

This is a code completion question. Please fill in the **EIGHT** blanks in the given source code to make it functional and provide the final output for the given test case.

```python
"""
This program counts the frequency of each word in a given text.
The function should be case-insensitive and should ignore common punctuation.
"""
import string

def count_word_frequency(text):
    # 1. Convert text to lowercase to ensure case-insensitivity
    text = 1._______________(3 points)

    # 2. Remove all punctuation marks from the text
    for punc in string.punctuation:
        text = text.replace(punc, " ")

    # 3. Split the text into a list of individual words
    words = 2._______________(3 points)

    # 4. Use a dictionary to count the frequency of each word
    word_counts = {}
    for word in words:
        if word:  # Make sure the word is not an empty string
            current_count = 3._______________(4 points)
            word_counts[word] = current_count + 1

    return word_counts
```

```python
def find_most_frequent(word_counts):
    if not word_counts:
        return None, 0

    most_frequent_word = ""
    max_count = -1

    # 5. Iterate through the dictionary to find the word with the highest count
    for word, count in 4._______________(4 points):
        if count > max_count:
            5._______________(2 points) = count
            6._______________(2 points) = word

    return most_frequent_word, max_count

def main():
    test_case = "Go, team, go! We must, we must, we will win!"

    frequencies = 7._______________(3 points)
    word, count = find_most_frequent(frequencies)
    print(f"Most frequent word: '{word}', Count: {count}")

if __name__ == "__main__":
    main()

# Provide the final printed output of the program:
# 8. ____________________________________(4 points)
```