Contents

1 Lecture 01 (Expressions)

1.1 Introduction to CS1010E

Problem Solving
- Formulate problem
- Think about solutions
- Express a solution clearly & accurately

Algorithm – conceretization of a solution to a problem
- well-defined computational procedure consisting of *a set of instructions*, that takes some value or set of values as *input*, and produces some value or set of values as *output*

Algorithm
- ideas
- machine *independent*

Program
- final code on a machine in a language
- machine *dependent*

Before coding, conceptualize the program by developing an **algorithm**
- algorithm usually expressed as
  - **Pseudocode** – artificial & informal language that helps programmers develop algorithms
    - "text-based"
  - **Flow Chart**
    - has different bubble types to signify different processes

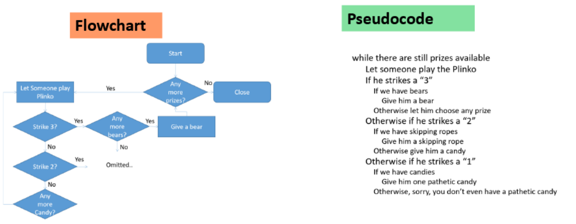


Figure 1: Example of a Flowchart & Pseudocode

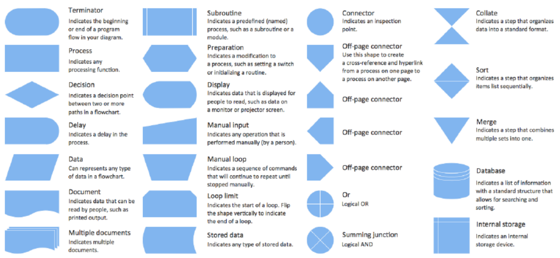


Figure 2: Flowchart Bubble Types

1.2 Python Programming Environment

Python – program file name has extension `.py`
- clear & readable syntax (text)
- intuitive
- natural expression
- powerful

IDE – Integrated Development Environment
- framework to *edit (write)* your program, *run (execute)* your program & *debug (correct)* the program
- different IDEs for different languages

e.g.

IDLE – Integrated Development and Learning Environment
- two windows[1]
  1. **editor** – contains your program
  2. **shell/console** – shows the execution of your program
     - input
     - output

1.3 Python Expressions

Python console/shell – runs a *Python interpreter* in the background
- **REPL**(oop)
  1. the interpreter **R**eads in user input
  2. **E**valuates it to some resulting value
  3. **P**rints out the result
  4. **L**oop

Values – basic entities that a program works with

e.g.

```
2
42
16.0
'Hello, World!'
```
(1)

Types[2] – classification; the class a value belongs to

e.g.

`2`	**integer**	
`16.0`	**floating-point number (real number)**	(2)
`'Hello, World!'`	**string**	

* *A value has (belongs to) a specific type*

e.g.

```
>>> type(2)
<class 'int'>
>>> type(16.0)
<class 'float'>
>>> type('Hello World!')
<class 'str'>
```

```
## 2 is different from '2'
>>> type ('2')
<class 'str'>
## 16.0 is different from '16.0'
>>> type('16.0')
<class 'str'>
## Do not use commas in a big number
>>> 1,000,000
(1, 0, 0)
```

[1]You develop your Python programs in the IDLE editor, not the IDLE shell/console
[2]Python tries to be aware of the types of the values it is working on

Variable – name that holds (refers to) a value

- **Variable Names**

  e.g.

  `my_name`, `thisFellow`

  - can be as long as you like
  - can only contain *letters*, *numbers*, and the *underscore character* '_'
  - cannot begin with a number
  - cAsE sEnsitiVE
  - camel form

    e.g.

    `newString`
  - snake form

    e.g.

    `a_string`
- **Assignment Statement** – associates a variable with a value through the equality symbol '='

  e.g.

```
message = 'May day! May day!'
n = 19
pi = 3.141592653589793
```
(3)

Keywords

- used by the interpreter to recognize the syntactic structure of a program
- displayed in different colors in the IDE
- cannot be used as variable names

  i.e.

```
>>> class = 'CS1010E'
SyntaxError: invalid syntax
```

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Figure 3: Python Keywords

Expressions – evaluated by the interpreter; the interpreter computes the value of an expression

- can be a *value*, a *variable*, or a *combination of values, variables & operators*

  e.g.

```
>>> 42
42
>>> n
19
>>> n + 25
44
```
- follows a precedence of operations (**PEDMAS**):[3]
  1. **P**arentheses
  2. **E**xponentiation
  3. **M**ultiplication & **D**ivision
  4. **A**ddition & **S**ubtraction

  e.g.

```
>>> 2 * (3 - 1)
4
>>> (1 + 1) * (5 - 2)
6
>>> 1 + 2 ** 3
9
```

```
>>> 2 * 3 ** 2
18
>>> -1 + 2 * 3
5
>>> 6 + 4 / 2
8.0
```

```
>>> 180 / 3 / 4
15.0
```

[3]Left associativity (except for exponentiation)

- **String Operations**
  - string concatenation

    e.g.

```
>>> '2' - '1'
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>> 'numerator' / 'denominator'
TypeError: unsupported operand type(s) for /: 'str' and 'str'
>>> 'A' * 'is born'
TypeError: can't multiply sequence by non-int of type 'str'
```

```
>>> first = 'thumbs'
>>> second = 'up'
## string concatenation
>>> first + second
'thumbsup'
>>> cheer = first + second
## repeat 3 times and concatenate
>>> cheer * 3
'thumbsupthumbsupthumbsup'
```

Statement – unit of code that has an effect
- executed by the interpreter; the interpreter does whatever the statement says
- does not necessarily result in a value

e.g.

Creating a variable or displaying a value

e.g.

```
>>> n = 29
>>> print(n)
29
```

Comments – annotations to help the programmer understand the intended logic
- ignored by the interpreter
- comment lines – starts with '`#`'
- multi-line string[4] – enclosed in '`'''...'''`'
  - creates a string in memory but doesn't assign it to anything

1.4 Exercises

1. Why is it called "Algorithm"?

   It was first written in a language called ALGO
   It was originated from the name of a person
   It emphasizes on the rhythmic aspect of computational procedures
   "Algo" refers to pseudocodes, "rithm" refers to flow charts
   None of the above

2. What does this program do?

```
a = 1
b = 2
c = a + b
if c < 0:
    print('c < 0')
else:
    print('c > 0')
```

   Print out `'c < 0'`
   Print out `'c > 0'`
   Print out `'#$!%'`
   Force pc to shut down

3. We develop our python programs in

   IDLE Console
   IDLE Editor
   IDLE Shell
   All of the above

[4] Not really a comment

4. The file name of a python program has the extension . . .

   .python
   .code
   .py
   .pptx
   None of the above

5. Which of the following is an expression?

   `234 ** 2 ** (-2)`
   `3.14159` – value
   `x = 3 + 432` – assignment statement
   `True` – boolean value
   `sqrt(3 + 2)`
   `print('Hello World!')` – statement
   `forward(300)` – command
   `1 <= 5 <= 3` – expression

6. Which of the following is a value?

   `True or not True` – expression
   `x` – variable name; expression
   `3.14159`
   `x = 3` – assignment statement
   `3 ** 2`
   `'Please choose me'` – string
   `22`
   `'22'` – string
   `left(30)` – statement

7. Which of the following is a variable name?

   `donaldTrump`
   `_hidden_name_`
   `x234`
   `2ndName` – starts with '2'; cannot start with a number
   `NUS_FoE`
   `False` – boolean value
   `Two*Three` – '*' is not allowed
   `amount$` – '$' is not allowed

1.5 Tutorial: Evaluation, Variables and Turtle

Python Shell

- echoes – "return values" of the previous line
  - some will not have any echo because it has no return value

Logical Evaluation

- Python has keywords: `True` & `False`
  - In Python 3.x

$$\left.\begin{array}{l}\texttt{True}\text{ will be equal to }1\\ \texttt{False}\text{ will be equal to }0\end{array}\right\} \qquad (4)$$

  * *anything that is not 0 or `None` will be evaluated as `True`*

e.g.

```
>>> 1 == 1
True
>>> 3 + 2 != 1 + 4
False
>>> 4 > 3
True
>>> 6 + 3 < 9 + 3
True
>>> True and (False or True)
True
>>> not True
False
>>> 0 and 9999
False
```

```
>>> not 'abc'
False
>>> not ''
True
>>> True and 0
0
>>> 1 or 0
1
>>> True + 1
2
>>> False * 5
False
>>> 0 + (not 1)
0
```

String Evaluation

- string indexing & slicing – `<string>[<start>:<stop>:<step>]`
  - `<stop>` – non-inclusive

  By default,
  - `<start>` – starts from index 0
  - `<stop>` – ends at the last letter; includes the last letter
  - `<step>` – increments by 1 step

  e.g.
  ```
  >>> 3 * 'gala'
  'galagalagala'
  ```
  ```
  >>> 'banana'[1::2]
  'aaa'
  ```
  e.g.
  ```
  >>> s = 'abcdef'
  >>> s[2]
  'c'
  >>> s[2:]
  'cdef'
  >>> s[2::]
  'cdef'
  >>> s[1:1]
  ''
  ```
  ```
  >>> s[]
  SyntaxError: invalid syntax
  >>> s[:2]
  'ab'
  >>> s[:2:]
  'ab'
  ```
- By Python convention if `<step>` is negative, default `<start>` is the last letter and default `<stop>` is the first letter inclusive

  e.g.
  ```
  >>> s[5:0:-1]
  'fedcb'
  ```
  ```
  >>> s[:2:-1]
  'fed'
  ```
  ```
  >>> s[::-1]
  'fedcba'
  ```
- **Lexicographical Order** – comparing letter-by-letter from left to right based on ASCII Table

  e.g.

  'K' > 'D';   'p' > 'P'

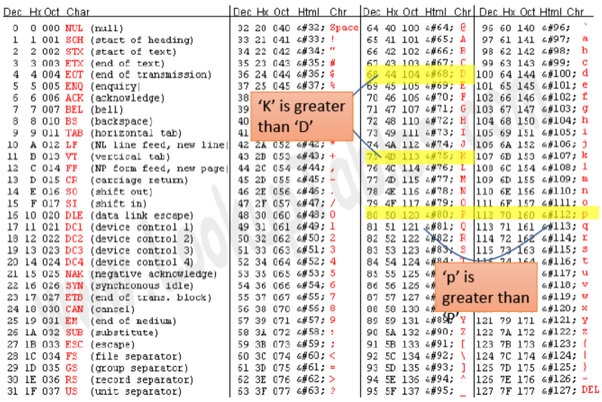


Figure 4: ASCII Table

  - "winner" is decided when the comparison reaches a different letter with a larger decimal value
  - "anything" is greater than "nothing"

  e.g.
  ```
  >>> 'abc' > 'abd'
  False
  >>> 'abc' > 'aba'
  True
  ```
  ```
  >>> 'ab' > 'aaabbbcccddd'
  True
  >>> 'abcd' > 'abc'
  True
  ```

Operator Precedence

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'td>
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Figure 5: Operator Precedence

Type Conversions

i.e.

```
>>> 1 == '1'
False
```

- '+' operator performs differently for different types

  e.g.

  ```
  >>> type(123)
  <class 'int'>
  >>> 123 + 456
  579
  ```

  ```
  >>> type('123')
  <class 'str'>
  >>> '123' + '456'
  '123456'
  >>> '123' + 456
  TypeError: can only concatenate str
  ↪  (not "int") to str
  ```

- **Float → Integer Conversion**

  i.e.

  ```
  >>> int(1.234)
  1
  ```

  ```
  >>> int(1.7)
  1
  ```

  - `int(...)` truncates the float; the integer produced is not always smaller than the input float

    e.g.

    ```
    >>> int(-2.3)
    -2
    ```

    ```
    >>> int(2.3)
    2
    ```

Variables

i.e.

```
>>> x = 3
>>> y = 4
>>> z = 5
>>> x * y + z
17
```

- can store any type

  e.g.

  ```
  >>> 5 > 3
  True
  >>> x = 5 > 3
  >>> x
  True
  ```

  ```
  >>> 5 > 3 and 3 > 9
  False
  ```

- performing operations on undeclared ("not created") variables will result in an error!
  - have to assign a value to the variable first before using it

Turtle Graphics

i.e.

```
from turtle import *
forward(100)
```

```
import turtle
turtle.forward(100)
```

- short forms of turtle functions exist

  e.g.

  `turtle.forward(...)` v.s. `turtle.fd(...)`

2 Lecture 02 (Functions)

2.1 Functions

Function – "takes" some *argument(s)* and "returns" some *result(s)*

i.e.

```
>>> <Function name>(<Function argument>)
<Return value>
```

- result is also called the `<Return value>`
- `<Function argument>` can also be an expression; can pass an expression to a function

  e.g.

  ```
  >>> h(3)
  4
  ```

  ```
  >>> h(15 * 60 + 30)
  2785
  ```

  ```
  >>> type(32)
  <class 'int'>
  ```

e.g.

```
>>> int('42')
42
>>> int('Hello')
ValueError: invalid literal for int()
↪ with base 10: 'Hello'
## no round off is performed
>>> int(3.999)
3
## just take the integer part
>>> int(-2.99)
-2
```

```
## convert an integer to a float
>>> float(32)
32.0
>>> float('2.33')
2.33
## convert its argument to string
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
## argument can be an expression
>>> str(1 + 2 + 3 + 4)
'10'
```

Categories of Python Functions

- **Built-in Functions** – already available when Python shell is started; need not be imported

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Figure 6: Python's built-in functions

- **Library Functions** – need to import from some packages/libraries/modules
  - **Module** – file that contains a collection of related functions

    e.g.
    - `math` – provides most familiar mathematical functions

      e.g.
      ```
      >>> import math
      >>> math
      <module 'math' (built-in)>
      >>> math.sin(math.pi / 2)
      1.0
      >>> math.sqrt(3 ** 2 + 4 ** 2)
      5.0
      ```
    - `pandas`
    - `numpy`
  - dot notation '.'
  - can be imported using different ways

    e.g.

    ```
    import math
    ```
    - more relaxed in using any functions in the whole library
    - require more memory to locate the library
    - needs dot notation & function name when calling function

    e.g.
    ```
    >>> import math
    >>> print(math.pi)
    3.141592653589793
    >>> print(math.cos(math.pi))
    -1.0
    ```
    ↪ makes code longer

    ```
    from math import cos
    ```
    - need to know exactly which function you have to import
    - saves memory
    - no need for dot notation when calling function; allows you to use function as it is

    e.g.
    ```
    >>> from math import pi, cos
    >>> print(cos(pi))
    -1.0
    ```
    ↪ makes code shorter

    ```
    from math import *
    ```
    - imports all functions available in the module
    - does not save memory
    - no need for dot notation when calling function; allows you to use function as it is

    e.g.
    ```
    >>> from math import *
    >>> print(pi)
    3.141592653589793
    >>> print(cos(pi))
    -1.0
    >>> print(tan(pi + e))
    -0.45054953406980836
    ```
    ↪ makes code shorter

- **User-defined Functions** – functions created by programmers
  - ↪ **Function Definition**
    - specifies the name of a new function
    - specifies sequence of statements that run when the function is called

    i.e.

    ```
    from math import sqrt
    def pythagoras(a, b):
        c2 = a ** 2 + b ** 2
        return sqrt(c2)
    ```

    ```
    >>> type(pythagoras)
    <class 'function'>
    >>> pythagoras(3, 4)
    5.0
    ```

    - starts with '`def`'
    - followed by *function name*
    - then *parameters* enclosed in brackets '`(...)`'
    - then '`:`'
      - to the left of '`:`' is *function header*
      - indented lines to the right of '`:`' is *function body*
    - indentation – denotes *function body*
  - **Value-returned Functions**
    - the last executable statement in the value-returned function definition is always a `return` statement
    - treat call to a value-returned function as an expression; assign the function call result to some variable

    e.g.

    ```
    def distance(x1, y1, x2, y2):
        hdist = (x2 - x1) ** 2
        vdist = (y2 - y1) ** 2
        dist = sqrt(hdist + vdist)
        return dist
    ```

    ```
    >>> distance(3, -1, 2, 3)
    4.123105625617661
    ```

    - **Leftmost-Innermost Evaluation**

      i.e.

      ```
      call(f(b(d(...), e(...)), a(...), c(...)))
      ```

      - a call to `f(...)` will evaluate the leftmost argument first – `b(...)`
      - calling `b(...)` will evaluate the subsequent leftmost argument – `d(...)`
      - then it evaluates the next argument in `b(...)` – `e(...)`
      - ...
  - **Void Functions** – performs an action without returning a value
    - no `return` statement in the function body
    - assigning the "result" of the call to a void function assigns a special value `None`

    e.g.

    ```
    def print_CS():
        print('To iterate is human,')
        print('To recurse, divine.')
    ```

    ```
    >>> type(print_CS)
    <class 'function'>
    >>> result = print_CS()
    To iterate is human,
    To recurse is divine.
    >>> print(result)
    None
    ```

Composition – taking small building blocks and composing them together

- **Using Functions** – have to create a function before you can run it
  - function definition has to be accepted by Python interpreter before the function gets called upon
- **Function Core Composition** – using the result of a function call as an argument in another function call

  e.g.

```python
def balance(principal, interest):
    return principal * (1 + interest)

def two_yr_compound(principal, interest):
    return balance(balance(principal, interest), interest)
```

```
>>> balance(10000, 0.05)
10500.0
>>> two_yr_compound(10000, 0.05)
11025.0
```

Running Python

- **Interactive Mode** – direct interaction with the Python interpreter
- **Script Mode** – saves code in a file (script) with the extension `.py` before loading the script to the Python interpreter for execution

Receiving Inputs to a Function

i.e.

`input(...)` – allows the user to interact with the Python execution in the shell

e.g.

```python
from math import sqrt
def solve_qe2():
    a = float(input('Coefficient a: '))
    b = float(input('Coefficient b: '))
    c = float(input('Coefficient c: '))
    delta = b ** 2 - 4 * a * c
    ans = (-b + sqrt(delta)) / (2 * a)
    ans = (-b - sqrt(delta)) / (2 * a)
    print("The two solutions are " + str(ans1) + " and " + str(ans2))
```

```
>>> solve_qe2()
Coefficient a: 1
Coefficient b: 5
Coefficient c: 6
The two solutions are -2.0 and -3.0
```

Flow of Execution

- execution begins at the first statement of the program
- statements run in the order from top down to bottom
- statements inside the function don't run until that function is called
- a function call makes a detour in the (top-down) flow of execution to the function block

Parameters & Arguments

- different functions may have different number of parameters

  e.g.

```
math.pow(2,3),
math.log10(10),
print_CS()
```

(5)

- arguments are assigned to function parameters during function application
- **parameter** – a variable used in function scope
- **argument(s)** – variable(s)/value(s) passed to function call
  - evaluated before the function is called

  * *arguments can be complex expressions*

e.g.

```
def print_2times(tony):
    print(tony)
    print(tony)
```

```
>>> print_2times('Love')
Love
Love
```

```
>>> print_2times(42)
42
42
>>> print_2times('care ' * 4)
care care care care
care care care care
```

Scoping

- variables created inside a function are **local** – it only exists inside the function
- parameters are also local

e.g.

```
def key_2times(high, low):
    key = high + low
    print_2times(key)
```

```
>>> training1 = 'HIGH KEY'
>>> training2 = 'low key'
>>> key_2times(training1, training2)
HIGH KEYlow key
HIGH KEYlow key
>>> print(key)
NameError: name 'key' is not defined
```

  - variables `high` & `low` do not exist outside `key_2times`

Stacking – stacked function calls

- when a function calls another function, the contents of the called function is stacked on the preceding function and executed first

2.2 Exercises

1. `>>> var = 29 * 24 - 39 / 23 + 94`

   This is an expression
   This is a statement – assignment statement; does not return a result
   This is both an expression and a statement
   None of the above

2. `>>> x + y = 34 * 50 + (49 - 23 * 10)`

   This is an expression
   This is a statement
   This is both an expression and a statement
   None of the above – syntax error

3. What is the output from the call `twice(twice(twice(2)))`?

```python
def twice(x):
    return x + x
```

`8`
`16`
`twice(2) + twice(2)`
`Error`
None of the above

4. What is the output from the call `strange(3, 100)`?

```python
def strange(a, b):
    c = a * b // b
    return c
    return c + 1
    return c + 2
```

`3`
`4`
`5`
`Error`
None of the above

5. What value will be returned from `f1(1, 2)`?

```python
def f3(x, y):
    return x

def f2(x, y):
    return f3(y, x)

def f1(x, y):
    return f2(y, x)
```

`1`
`2`
`Error: parameters name clash`

6. What value will be returned from `f1(1, 2)`?

```python
def g(a, b):
    (a + b) * 2

def f(x, y):
    res = g(x, y)
    return res + res
```

`4`
`6`
`8`
`Error`
None of the above

As there is no return statement in the function definition of `g`, `None` will be assigned to `res` when `g` is called in `f`. Performing addition on `None` will give an `Error`.

2.3 Tutorial: Functions

Rounding

i.e.

```python
round(..., x)
```

– rounds answer to x decimal places using the round function

e.g.

```python
>>> round(1 / 0.6214, 3)
1.609
```

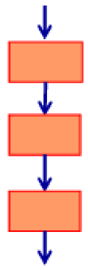
General guidelines/strategies to approaching computing problems

1. Take note of important information
   – check if you need to convert any information into a more useful form
2. List down ideas and check their feasibility
   – choose the one you have the most confidence in
3. Look at tools picked up in lectures/tutorials/learning
   – apply the most useful one
4. Create test cases
   – check if it is correct; revert to Step 2 if there is a major error

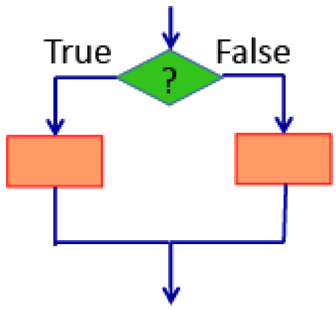
3 Lecture 03 (Conditionals & Iterations)

Control Structures

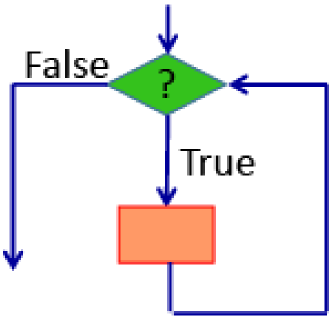
- **Sequence** – default
  - top to bottom
  - statement by statement



- **Selection** – "branching"



- **Repetition** – "looping"



3.1 Conditionals

Control Structure: Selection

i.e.

```
if <expr>:
    <if statement>
else:
    <else statement>
```

- *<expr>* – "if" test
  - value of type `Bool`
  - conditional expression/test; relation
    - if the condition evaluates to `True`, the *<if statement>* will run;
    - if the condition evaluates to `False`, *<else statement>* will run
- *<if statement>* – "then" branch
  - indented
  - can be more than one statement
- *<else statement>* – "else" branch
  - indented
  - can be more than one statement
  - may be omitted

e.g.

```python
delta = b ** 2 - 4 * a * c
if delta >= 0:
    ans1 = (-b + sqrt(delta)) / (2 * a)
    ans1 = (-b - sqrt(delta)) / (2 * a)
    print("The two solutions are " +
    ↪   str(ans1) + " and " +
    ↪   str(ans2))
else:
    print("The equation has no real
    ↪   root")
```

- **Nested 'if'**

i.e.

```
if <expr>:
    if <expr>:
        <statement(s)>
```

e.g.

```python
a = 3
if a < 10:
    if a < 4:
        print('Here')
```

```
>>> my_account = 1000
>>> if my_account < 0:
        print('poor')
    else:
        if my_account >
        ↪   100:
            print('v
            ↪   rich')
v rich
```

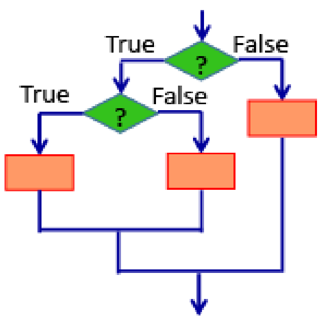


Figure 7: Nested 'if'

- **Conditional '`elif`'**
  - can be many

i.e.

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
else:
    <statement(s)>
```

e.g.

```python
def grade(score):
    if score >= 90:
        return 'A+'
    elif score >= 85:
        return 'A'
    elif score >= 80:
        return 'A-'
    elif score >= 75:
        return 'B+'
    elif score >= 70:
        return 'B'
    elif score >= 65:
        return 'B-'
    elif score >= 60:
        return 'C+'
    elif score >= 55:
        return 'C'
    elif score >= 50:
        return 'C-'
    elif score >= 45:
        return 'D+'
    elif score >= 40:
        return 'D'
    else:
        return 'F'
```

```
>>> grade(43)
'D'
>>> grade(57)
'C'
```

3.2 Repetition / Iteration

Iteration – the act of *repeating a process* with the aim of *approaching a desired goal*, target or result

- ***Infinite Repetition*** – to avoid, check that updating loop indexes will eventually lead to the terminating condition – loop condition evaluating to `False`
  ↪ Well-founded relation

`while` Loop

e.g.

```python
while n > 0:
    print(n)
    n = n - 1
```

- starts with '`while`'
- followed by *loop condition*
  - value of type `Bool`
  - if the condition evaluates to `True`, the loop body executes before checking the loop condition again
  - if the condition evaluates to `False`, the `while` loop is skipped and proceeds to the next statement outside the `while` loop
- then '`:`'
  - indented lines to the right of '`:`' is *loop body*

for Loop

i.e.

```
for <var> in <sequence>:
    <body>
```

- <var> – variable that takes each value in the sequence
- <sequence> – sequence of values
- <body> – statement(s) that will evaluated for each value in the sequence

- iterating over a range of numbers
  - range function

    i.e.

    ```
    range(<start>, <stop>, <step>)
    ```

    - creates a sequence of integers
      - from <start> (inclusive) to <stop> (non-inclusive); semi-closed interval – [<start>, <stop>)
      - incremented by step

    * <start> & <step> may be omitted

    e.g.

    - up to, but not including, 5

      i.e.

      ```
      >>> for i in range(5):
              print(i)
      0
      1
      2
      3
      4
      ```

    - up to, but not including, 10

      i.e.

      ```
      >>> for i in range(1, 10, 3):
              print(i)
      1
      4
      7
      ```

    - up to, but not including 6

      i.e.

      ```
      >>> for i in range(3, 6):
              print(i)
      3
      4
      5
      ```

    - down to, but not including, 5

      i.e.

      ```
      >>> for a in range(10, 5, -2):
              print(a)
      10
      8
      6
      ```

- iterating over a string

  e.g.

  ```
  >>> emotion = 'happy'
  ```

  ```
  >>> for i in range(len(emotion)):
          print(emotion[i])
  h
  a
  p
  p
  y
  ```

  ```
  >>> for c in emotion:
          print(c)
  h
  a
  p
  p
  y
  ```

Loop Invariant – state property that always hold at the beginning of any iteration in a loop

Types of Loops

1. Must run exactly c times, for some constant c
   - has a(n) loop index/index variable
     - reassignment of loop index controls the number of the iterations the loop body is to be executed

   e.g.

   Computing factorial

```
def factorial(n):
    product = 1
    for counter in range(2, n + 1):
        ## 2 <= counter <= n &
        ## product = 1 * 2 * ... * (counter - 1)
        product = product * counter
    return product
```

   Flipping coins

```
import random
def flipCoins():
    print('I will flip a coin 1000 times.')
    print('Guess how many times it will come up heads.')
    heads = 0
    for flip in range(0, 1000):
        ## invariant: 0 <= heads <= flip
        if random.randint(0, 1) == 1:
            heads = heads + 1
    print(heads)
```

2. Run some number of times, not known beforehand
   - accumulator – variable declared outside of loop scope such that assigned value persists in the next iteration
     - also used in loop condition to determine terminating condition
     - often starts from `0`

       ↪ `0` – additive identity; unit of addition operators

   e.g.

   Function `sumNumbers()` that continues to read numbers from a user and sums up the numbers when the user enters the word "`bye`"

```
def sumNumbers():
    sumSoFar = 0
    print('Enter a number or type \'bye\' to sum: ')
    num = input()
    while num != 'bye':
        ## inside while loop -- value of num is not bye
        sumSoFar = sumSoFar + int(num)
        print('Enter a number or type \'bye\' to sum: ')
        num = input()
    ## Out of the while loop -- num has value 'bye'
    print('The sum of all numbers is ' + str(sumSoFar))
```

Guess a number game

```python
def guessANum():
    ## 0 <= secret <= 99
    secret = random.randint(0, 99)
    guess = -1
    print('I have a number in mind between 0 and 99')
    while guess != secret:
        guess = int(input('Guess a number: '))
        if guess == secret:
            print('Bingo!!! You got it! ')
        elif guess < secret:
            print('Your number is too small')
        else:
            print('Your number is too big')
```

```
>>> sumNumbers()
Enter a number or type 'bye' to
↪ sum:
10
Enter a number or type 'bye' to
↪ sum:
39
Enter a number or type 'bye' to
↪ sum:
-20
Enter a number or type 'bye' to
↪ sum:
4
Enter a number or type 'bye' to
↪ sum:
bye
The sum of all numbers is 33
```

```
>>> guessANum()
I have a number in mind between 0 and
↪ 99
Guess a number: 50
Your number is too big
Guess a number: 25
Your number is too big
Guess a number: 12
Your number is too big
Guess a number: 6
Your number is too small
Guess a number: 9
Your number is too big
Guess a number: 7
Bingo!!! You got it!
```

3. Run at most c times, for some constant c
   - **Universality Check** – check all `True` (or check all `False`)

     e.g.

     Checks if all characters are alphabets

```python
def checkAllAlpha(string):
    for i in range(len(string)):
        ## string[0] ... string[i - 1] are all alphabet
        if not string[i].isalpha():
            return False
    return True
```

– **Existentiality Check** – find any `True` (or `False`)

e.g.

Checks if there is any non-alphabet in a character string

$$\exists a \in string : \neg alphabet(a) \equiv \neg(\forall a \in string : alphabet(a)) \tag{6}$$

```
def someNotAlpha(string):
    for char in string:
        if not char.isalpha():
            return True
    return False
```

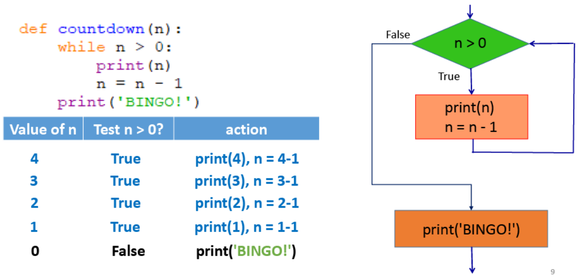


Figure 8: Example of a `while` loop that runs exactly c times

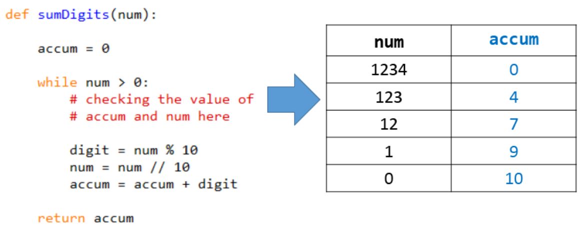


Figure 9: Example of a `while` loop that runs an unknown number of times

`break` – breaks out of a loop

– will only break out from the innermost loop

`continue` – continues to next iteration

3.3 Exercises

1. Given three distinct integers `x`, `y`, `z`, return the value which is in the middle, not the biggest, not the smallest.

```
def middle(x, y, z):
    ...expression...
```

2. What printed out from `f(2)`, `f(5)`, `f(15)`?

```
def f(a)
    if a < 10:
        if a < 4:
            print('Here')
        else:
            print('There')
    else:
        print('Where')
```

`'Here', 'There', 'Where'`

`'Here', 'Where', 'There'`

`'There', 'Here', 'Where'`

`'Where', 'There', 'Here'`

None of the above

3. What printed out from executing `g(2)`?

```
def g(a):
    if a < 10:
        if a < 4:
            print('Here')
        else:
            print('There')
    print('Where')
```

- 'Here'
- 'Where'
- 'Here' and 'Where'
- 'There' and 'Where'
- None of the above

4. What printed out when calling `pain()`?

```
def pain():
    if False:
        print('False')
    else:
        if True:
            print('True')
        else:
            print('neither True nor False')
```

- True
- False
- Neither True nor False
- None
- None of the above

5. Which of the argument value will `chk1` and `chk2` print identically?

```
def chk1(a):
    if a > 10:
        print('yes')
    print('no')

def chk2(a):
    if a > 10:
        print('yes')
    else:
        print('no')
```

- 5
- 10
- 15
- 20

6. Which of the argument value will calls to `chk3` and `chk4` return identical results?

```
def chk3(a):
    if a > 10:
        return 'yes'
    return 'no'

def chk4(a):
    if a > 10:
        return 'yes'
    else:
        return 'no'
```

- 5
- 10
- 15
- 20

* *Return is the last statement to be executed*

3.4 Tutorial: Selection and Repetitions

`print` Function

- `print(..., end = '')`
  - forces the `print` function to end with an empty string (`''`) instead of a new line

random Library Functions

- returns an *integer*
  - random.randint(..., ...)

    e.g.
    ```
    random.randint(0, 1000)
    ```
  - random.randrange(..., ...)

    e.g.
    ```
    random.randrange(0, 1001)
    ```
- returns a *floating point number*
  - random.random()

    e.g.
    ```
    random.random() * 1000
    ```
  - random.uniform(..., ...)

    e.g.
    ```
    random.uniform(0, 1000)
    ```

4 Lecture 04 (Lists)

4.1 Sequences

Categories of Data

- **Primitive**
  - available in every programming language

  e.g.
  - Integers
  - Floating-point Numbers
  - Booleans
- **Structural / Compound Data**

  e.g.
  - Rational number of 2 integers: $\frac{m}{n}$
  - Student record containing: name, ID, grades
  - Sequences
  - Objects

Sequence – a collection of *'something'*

- Types of sequences in Python:
  - Strings – sequences of characters

    e.g.
    ```
    >>> name = 'khooSC'
    >>> course_code = 'CS1010E'
    >>> course_code
    'CS1010E'
    >>> course_code[2]
    '1'
    ```
  - Lists – sequences of anything; can contain values from more than one type

    i.e.
    ```
    [..., ..., ..., ...]
    ```
    - enclosed in square brackets
    - separated by commas
  - Tuples
  - Others:
    - Sets
    - Dictionary

Operations on Indexed Sequences

- **Strings**

  e.g.
  ```
  s1 = 'Minions like bananas'
  ```
  - <*string*>[i] – returns the i^{th} element

    e.g.
    ```
    >>> s1[5]
    'n'
    ```
  - <*string*>[i:j] – returns elements i up to j - 1; [i, j)

    e.g.
    ```
    >>> s1[0:6]
    'Minion'
    ```
  - len(<*string*>) – returns the number of elements

    e.g.
    ```
    >>> len(s1)
    21
    ```

- min(<*string*>) – returns the smallest value

  e.g.

  ```
  >>> max(s1)
  's'
  ```

- max(<*string*>) – returns the largest value

  e.g.

  ```
  >>> min(s1)
  ' '
  ```

- <*substring*> in <*string*>
  - returns True if <*substring*> is a part of <*string*>

  e.g.

  ```
  >>> 'o' in s1
  True
  ```

  ```
  >>> 'z' in s1
  False
  ```

- <*substring*> not in <*string*>
  - returns True if <*substring*> is not a part of <*string*>
- <*string a*> + <*string b*>
  - returns a new sequence that concatenates <*string a*> & <*string b*>

  e.g.

  ```
  >>> s1 + ' and Gru'
  'Minions like bananas and Gru'
  ```

- n * <*string*>
  - returns a new sequence with n copies of <*string*>

  e.g.

  ```
  >>> s1 * 3
  'Minions like bananas Minions like bananas Minions like bananas'
  ```

e.g.

Check if the string is in ascending order

```
def ascending(a_string):
    if len(a_string) <= 1:
        return True
    for i in range(len(a_string) - 1):
        if a_string[i] > a_string[i + 1]:
            return False
    return True
```

Produce a new string in which all characters are m characters always

```
def increment(a_string, num):
    newString = ''  ## camel form
    for character in a_string:
        newChar = chr(ord(character) + num)
        newString = newString + newChar
    return newString
```

- **Lists**

  e.g.

  ```
  >>> even_numbers_10 = [0, 2, 4, 6, 8, 10]
  >>> my_good_friends = ['Peters', 'Paul', 'Mary']
  >>> ans_to_universe = ['Nothing', 'Deity', 42, True, None]
  >>> ans_to_universe[3:5]
  [True, None]
  >>> len(ans_to_universe)
  4
  >>> type(ans_to_universe)
  <class 'list'>
  >>> type(ans_to_universe[0])
  <class 'str'>
  >>> type(ans_to_universe[2])
  <class 'int'>
  >>> type(ans_to_universe[4])
  <class NoneType>
  ```

  - **Block (Pointer) Diagrams** – way to visualize list structures
    - assigning a list to `list1` & `list2`

      i.e.

      ```
      >>> list1 = [1, 2, 3]
      >>> list2 = ['a', 'b', 'c']
      ```

      ↪ `list1` & `list2` points to two different structures containing elements; points to two structures in memory

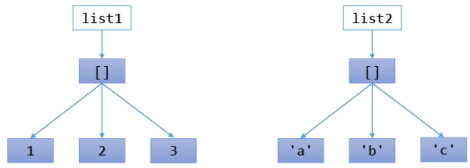


Figure 10: Assigning a list to a variable

    - assigning a list as an element to another list

      i.e.

      ```
      >>> list3 = [list1, list2]
      >>> list3
      [[1, 2, 3], ['a', 'b', 'c']]
      ```

      ↪ `list3` points to `list1` & `list2`

      i.e.

      changing `list1[1]` will change `list3[0][1]`

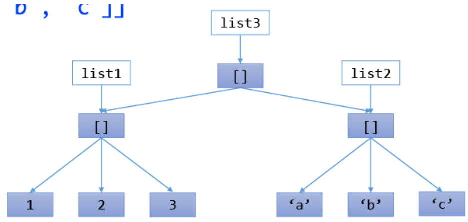


Figure 11: Assigning a list as an element to another list

    - **Concatenation**
      - joining two lists together using '+'

      i.e.

      ```
      >>> list3 = list1 + list2
      >>> list3
      [1, 2, 3, 'a', 'b', 'c']
      ```

      ↪ creates a new list in memory that doesn't point to other structures

      i.e.

      changing `list2[1]` will NOT change `list3[4]`

    - **Copy**
      - replicating a list

      i.e.

      ```
      >>> list3 = 2 * list1
      >>> list3
      [1, 2, 3, 1, 2, 3]
      ```

      ↪ creates a new list in memory that doesn't point to other structures

      i.e.

      changing `list1[1]` will NOT change `list3[1]` or `list[4]`

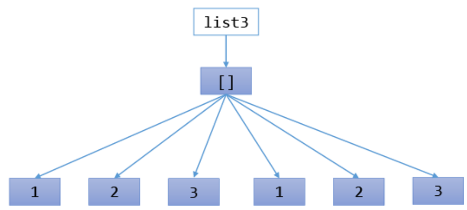


Figure 12: Copying a list

e.g.

```
>>> list4 = [True, list3, list1]
>>> list4
[True, [[1, 2, 3], ['a', 'b', 'c']], [1, 2, 3]]
>>> list5 = []
>>> list5
[]
```

* *[] – empty list / null list*

Mutability – ability to reassign (change) the value of an element

- lists are *mutable*

  e.g.

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

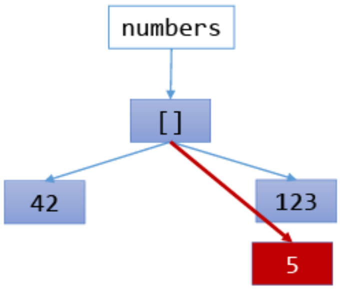


Figure 13: Reassigning the value of an element in a list

- strings are *immutable*

  e.g.

```
>>> str = 'abcdefg'
>>> str[1] = 'z'
TypeError: 'str' object does not support item assignment
```

Traversing Elements

- **Iterables** – right-side of a `for` loop header
  - anything that "can be looped over"
    - a string
    - a range
    - a list
- using `for` loop

  e.g.

```
>>> for num in numbers:
        print(num)
42
5
>>> numbers
[42, 5]
```

```
>>> for i in range(len(numbers)):
        numbers[i] *= 2
>>> numbers
[84, 10]
```

  - using a `for` loop over an empty list will never execute the loop body

    e.g.

```
>>> for x in []
        print('This never happens.')
```

- nested lists are counted as a single element when accessed in a `for` loop execution

  e.g.

```
>>> misc = ['spam', 1, [2, 3, 4, 5]]
>>> for i in range(len(misc)):
        print(i, misc[i])
0 spam
1 1
2 [2, 3, 4, 5]
```

Deleting Elements

- destructive operations – changes/modifies the list

e.g.

```
t = ['a', 'b', 'c', 'b']
```

- known index – `del <list>[<index>]`

  e.g.

```
>>> del t[1]
>>> t
['a', 'c', 'b']
>>> del t[4]
IndexError: list assignment index out of range
```

- known range – `del <list>[<start>:<stop>:<step>]`

  e.g.

```
>>> del t[1:4:2]
>>> t
['a', 'c']
```

- unknown index – `<list>.remove(...)`
  - removes the first instance

  e.g.

```
>>> t.remove('b')
>>> t
['a', 'c', 'b']
>>> t.remove('d')
ValueError: list.remove(x): x not
↪ in list
```

- delete and return deleted element of known index
  - `<list>.pop(<index>)`[5]

  e.g.

```
>>> x = t.pop(1)
>>> t
['a', 'c', 'b']
>>> x
'b'
```

[5]By default removes and returns the last element if <index> is left empty

`<list>.append(...)` – appends a new element to the *end* of a list

- always "extends" a list by one element
- void function
  ↪ assigning the result to a variable will assign `None`!

e.g.

```
>>> t = [1, 2, 3]
>>> t.append(4)
>>> t
[1, 2, 3, 4]
>>> t1 = [1, 2, 3]
>>> t2 = [4, 5]
>>> t1.append(t2)
>>> t1
[1, 2, 3, [4, 5]]
```

`<list>.extend(<sequence>)` – extends a list by joining a sequence at the end

- "appends" the elements of a sequence to the end of a list
- void function
  ↪ assigning the result to a variable will assign `None`!
- passing a value (string/integer/float) to `<sequence>` will result in an error
  e.g.

```
>>> lst2 = [1, 2, 3]
lst2.extend(7)
TypeError: 'int' object is not
↪  iterable
```

e.g.

```
>>> t1 = [1, 2, 3]
>>> t2 = [4, 5]
>>> t1.extend(t2)
>>> t1
>>> [1, 2, 3, 4, 5]
```

Examples

- **Linear Scan of a List**
  Given a list, `scanPlus` cumulates a list of intermediate cumulative results from beginning of a list using a start value.

```
def scanPlus(lst, id):
    acc = []
    prefix = id
    for item in lst:
        prefix = prefix + item
        acc.append(prefix)
    return acc
```

```
>>> aList = ['a', 'b', 'c', 'd', 'e']
>>> scanPlus(aList, 'Z')
['Za', 'Zab', 'Zabc', 'Zabcd',
↪  'Zabcde']
```

↪ prefix computation
e.g.

```
>>> alphalst = list('abcdefg')
>>> alphalst
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> scanPlus(alphalst, '')
['a', 'ab', 'abc', 'abcd', 'abcde', 'abcdef', 'abcdefg']
```

↪ prefix sum of a series
e.g.

```
>>> gpSeries = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
>>> scanPlus(gpSeries, 0)
[2, 6, 14, 30, 62, 126, 254, 510, 1022, 2046]
```

– **Generate Prime Numbers** < n – Sieve of Eratosthenes

```python
def genPrime(n):
    up2n = []
    for i in range(2, n):
        up2n.append(i)
    ## delete multiples of 2
    twos = 2 * 2
    while twos < n:
        up2n.remove(twos)
        twos += 2
    ## delete multiples of odds, starting from 3
    for oddN in range(3, n, 2):
        if oddN not in up2n:
            continue
        for odds in range(oddN * 2, n, oddN):
            if odds in up2n:
                up2n.remove(odds)
    return up2n
```

```
>>> genPrime(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
↪ 73, 79, 83, 89, 97]
>>> len(genPrime(120))
30
```

4.2 Call by Value v.s. Aliasing

How Python handles two names for the same thing in function scope and global scope

Call by Value (string/integer/float)

- Passing a *string/integer/float* as an argument to a function creates a copy of the *string/integer/float*
  ↪ the copied *string/integer/float* is passed over to the function parameter
- both the variable in the global scope and the parameter in the function scope will have a copy of the same *string/integer/float*

e.g.

```python
def exchange(x, y):
    print('Before exchange: ')
    print('x = ' + str(x) + '; y = ' + str(y))
    x, y = y, x
    print('After exchange: ')
    print('x = ' + str(x) + '; y = ' + str(y))
```

```
>>> n1 = 3
>>> n2 = 7
>>> n1, n2
(3, 7)
>>> exchange(n1, n2)
Before exchange:
x = 3; y = 7
After exchange:
x = 7; y = 3
>>> n1, n2
(3, 7)
```

```
>>> n1 = 'FoE'
>>> n2 = 'SoC'
>>> n1, n2
('FoE', 'SoC')
>>> exchange(n1, n2)
Before exchange:
x = FoE; y = SoC
After exchange:
x = SoC; y = FoE
>>> n1, n2
('FoE', 'SoC')
```

Call by Reference (list)

- Passing a list as an argument to a function copies the pointers to the list
  - ↪ the copied pointers are passed over to the function parameter
  - ↪ both the variable in the global scope and the parameter in the function scope are *aliases* for the same object
    - **Aliasing** – referencing the same piece of data in the memory; same address in memory
- allows for a variable inside function scope to mutate an alias variable outside function scope

e.g.

```python
def updateVal(val):
    print('Before: ' + str(val))
    val.append(30000)
    print('After: ' + str(val))
```

```
>>> lst = [1, 2]
>>> lst
[1, 2]
>>> updateVal(lst)
Before: [1, 2]
After: [1, 2, 30000]
>>> lst
[1, 2, 30000]
```

More Examples

- **Linear Scan of a List – In-place**

```python
def scanPlusInPlace(lst, id):
    prefix = id
    for i in range(len(lst)):
        prefix = prefix + lst[i]
        lst[i] = prefix
```

```
>>> aList = ['a', 'b', 'c', 'd', 'e']
>>> scanPlusInPlace(aList, 'Z')
>>> aList
['Za', 'Zab', 'Zabc', 'Zabcd', 'Zabcde']
```

4.3 Exercises

1. What wll be printed out?

```python
def mnmx(string):
    mm = string[0]
    MM = string[0]
    for i in range(len(string)):
        if string[i] < mm:
            mm = string[i]
        if string[i] > MM:
            MM = string[i]
    print(mm, MM)

mnmx('admistrationpanel')
```

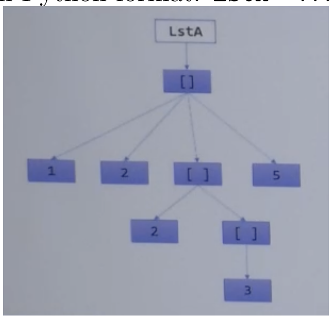
```
t a
a t
l a
a l
```
None of the above

2. What is the latest content of `lst`?

```
>>> lst
[-5, -3, -1, 1, 3]
>>> lst3
['a', 'b', 'c']
>>> len(lst[0:10:2])
3
>>> lst[0:10:2] = lst3
>>> lst
```

```
['a', -3, 'b', 1, 'c']
[-5, 'a', -1, 'b', 3, 'c']
[-5, -3, -1, 1, 3]
['a', 'b', 'c']
```

3. Express the block diagram below in Python format: `LstA` =???



`[1, 2, [2, 3], 5]`
`[1, 2, [2, [3]], 5]`
`[[1], [2], [2, 3], [5]]`
`[1, 2, [], 2, [], 3, 5]`

4. Given the following execution, what will be printed on the console?

```
>>> lst = [0] * 5
>>> lst[3] = 15
>>> print(lst)
```

`[0, 0, 0, 0, 0]`
`[0, 0, 0, 15, 0]`
`[15, 15, 15, 15, 15]`
Error during execution
None of the above

5. What will be printed after executing the following code?

```
>>> lst2 = [[0]] * 3
>>> lst2[1][0] = 30
>>> print(lst2)
```

`[[0], [30], [0]]`
`[[0], [0, 30], [0]]`
`[[30], [30], [30]]`
Error
None of the above

4.4 Tutorial: Lists

List Operations

e.g.

```
x = ['a', 'b', 'c', 'd']
```

- `len(<list>)` – returns the number of elements
  e.g.
  ```
  >>> print(len(x))
  4
  ```
- `<element> in <list>`
  - returns `True` if `<element>` is in `<list>`, and `False` otherwise

  e.g.
  ```
  >>> print('a' in x)
  True
  ```
  ```
  >>> print('f' in x)
  False
  ```
- `for <var> in <list>`
  - iterates over all the elements of `<list>`; each element stored in `<var>`
- `max(<list>)` – returns the maximum element
  e.g.
  ```
  >>> print(max(x))
  d
  ```
- `min(<list>)` – returns the minimum element
  e.g.
  ```
  >>> print(min(x))
  a
  ```

- `<list>.reverse()`
  - modifies `<list>` by reversing it
  - no return; void function

  e.g.
```
>>> lst.reverse()
>>> print(lst)
['d', 'c', 'b', 'a']
```

- `<list>.insert(<index>, <element>)`
  - inserts `<element>` at `<index>`

  e.g.
```
>>> lst.insert(2, 'e')
>>> print(lst)
['a', 'b', 'e', 'c', 'd']
```

- `<list>.pop()`
  - removes & returns the last element of `<list>`

  e.g.
```
>>> x = lst.pop()
>>> print(x)
d
>>> print(lst)
['a', 'b', 'c']
```

- `<list>.pop(<index>)`
  - removes & returns the element of `<list>` at index `<index>`

  e.g.
```
>>> x = lst.pop(2)
>>> print(x)
c
>>> print(lst)
['a', 'b', 'd']
```

- `<list>.remove(<element>)`
  - modifies `<list>` by removing the first occurrence of `<element>`
  - no return; void function

  e.g.
```
>>> lst.remove('a')
>>> print(lst)
['b', 'c', 'd']
```

- `<list>.clear()`
  - empties `<list>`

  e.g.
```
>>> lst.clear()
>>> print(lst)
[]
```

- `<list>.copy()`
  - returns a shallow copy of `<list>`

  e.g.
```
>>> x = lst.copy()
>>> print(x)
['a', 'b', 'c', 'd']
```

List Access

- 2 types of index for a list of size n
  - Forward index – starts from `0` ends at `n - 1`
  - Backward index – starts from `-1` ends at `-n`

0	1	2	3	4	5	6	7
e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
-8	-7	-6	-5	-4	-3	-2	-1

e.g.

```
>>> lst = [1, 2, 3, [4, 5], 6, 7]
>>> lst[2]
3
>>> lst[-2]
6
>>> lst[3]
[4, 5]
```

Ways to Avoid Call by Reference

- duplicating

  i.e.

  ```
  <new list> = <list>.copy()
  ```

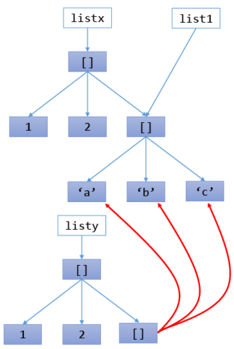
  e.g.

  ```
  >>> listx = [1, 2, 3]
  >>> listy = listx.copy()
  >>> listx[0] = 999
  >>> print(listy)
  [1, 2, 3]
  >>> print(listx)
  [999, 2, 3]
  ```

  * *only a **shallow** copy; only duplicates the first layer*

  e.g.

  ```
  >>> list1 = ['a', 'b', 'c']
  >>> listx = [1, 2, list1]
  >>> listy = listx.copy()
  >>> listy
  [1, 2, ['a', 'b', 'c']]
  >>> list1[0] = 'z'
  >>> listy
  [1, 2, ['z', 'b', 'c']]
  ```



- sequence slicing

  i.e.

  ```
  <new list> = <list>[:]
  ```

- typecasting

  i.e.

  ```
  <new list> = list(<list>)
  ```

5 Lecture 05 (Recursions)

5.1 Recursion

Recursion - paradigm of solving a big problem by solving some smaller versions of itself

- a function that calls itself
- *Divide-and-Conquer* technique

i.e.

```
def recursive(n):
    if <terminal condition>:
        return <base case>
    else:
        return <recursive case>
```

- `<terminal condition>` – condition for which when satisfied, the recursion will stop
- `<base case>` – Degenerated/Terminating case; does not call itself
- `<recursive case>` – Inductive case; calls itself
  - calls itself with reduced problem size at every deeper level; gets closer to the `<base case>`

e.g.

```python
def dream(level):
    print(' ' * level, 'Entered dream level ', str(level))
    if (level == 3):
        wakeup(level)
    else:
        dream(level + 1)
        wakeup(level)

def wakeup(level):
    print(' ' * level, 'Waking up from level ', str(level))
```

```
>>> dream(0)
 Entered dream level  0
  Entered dream level  1
   Entered dream level  2
    Entered dream level  3
    Waking up from level  3
   Waking up from level  2
  Waking up from level  1
 Waking up from level 0
```

- **Infinite Recursion**
  - if recursion *fails to* reach a base case, it will continue making recursive calls *forever*; program execution never terminates
    ↪ most likely leads to a crash[6]
  - Python tracks how many stacked calls there are and when it reaches a certain number of stacked calls, stops code execution with a `RecursionError` instead

  e.g.

```python
def recurse(n):
    if n > 0:
        return recurse(n + 1)
    return recurse(-n)
```

```
>>> recurse(10)
RecursionError: maximum recursion depth exceeded in comparison
```

Rules for Effective Recursion

- Developing a recursive solution to a problem
  1. Identify problems of smaller sizes
  2. Find the *pattern* that holds persistently in both smaller & bigger problems
  3. Work on simplest/smallest problem to get the solution
     ↪ extend solution to solve a bigger problem; use the solution for the smaller problem to solve the bigger problem
  4. Check if the pattern of using solutions for the smaller problem to solve the bigger problem is applicable to solve even bigger problems

[6]Python will not let the program crash

Examples

- **Factorial**
  - defined by

$$
\begin{aligned}
0! &= 1 \\
n! &= 1 \times 2 \times 3 \times \cdots \times (n-1) \times n, \qquad n > 0 \\
&= (1 \times 2 \times 3 \times \cdots \times (n-1)) \times n \\
&= (n-1)! \times n
\end{aligned} \tag{7}
$$

  - iterative

$$n! = 1 \times 2 \times 3 \times \cdots \times n \tag{8}$$

```python
def factorial(n):
    ans = 1
    i = 1
    while i <= n:
        ans = ans * i
        i = i + 1
        return ans
```

  - recursive

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{otherwise} \end{cases}$$

```python
def factorialR(n):
    if n == 0:
        return 1
    else:
        return factorialR(n
        ↪  - 1) * n
```

- **How Many Ways to Park Vehicles?**

  Let's say we have two types of vehicles, cars and buses. And each car can park into one parking space, but a bus needs two consecutive ones. If we have 1 parking space, we can only park a car. But if there are 2 parking spaces, we can either park a bus or two cars. In general if we have n parking spaces, how many ways can we park the vehicles?

  Let $f(n)$ be the function, which given n parking spaces, determines how many ways there are to park the vehicles.

  Think recursively by relying on the solutions to the smaller problems – If we park a car/bus in the last parking lot, how many ways can we park vehicles in the remaining lots?

  - $f(2)$
    - **car** $\rightarrow f(1)$ ways
    - **bus** $\rightarrow f(0)$ ways

    $\implies f(2) = f(1) + f(0)$
  - $f(3)$
    - **car** $\rightarrow f(2)$ ways
    - **bus** $\rightarrow f(1)$ ways

    $\implies f(3) = f(2) + f(1)$

  Hence, if the last parking lot has a **car**, there will be $n-1$ spaces left. If the last parking lot has a **bus**, there will be $n-2$ spaces left.

  $\therefore$ The number of ways to park vehicles at n spaces:

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ f(n-1) + f(n-2) & \text{if } n > 1 \end{cases} \tag{9}$$

```python
def f(n):
    if n <= 1:
        return 1
    return f(n - 1) + f(n -
    ↪  2)
```

```
>>> f(5)
8
>>> f(6)
13
>>> f(7)
21
```

- **Rabbit's Mating Problem**

  Rabbits are able to mate at the age of one month, and pregnancy takes one month. Thus, at the end of its second month, a female can produce another pair of rabbits. Rabbits never die, and a mating pair always produces one new pair (one male, one female) every month from the second month on. If we start with a new pair from birth, how many pairs will there be in one year?

		Old Rabbits		Newly born
$R(0)$	$=$			1
$R(1)$	$=$	1		
$R(2)$	$=$	1	$+$	1
$R(3)$	$=$	2	$+$	1
$R(4)$	$=$	3	$+$	2
$R(5)$	$=$	$R(4)$	$+$	$R(3)$

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ f(n-1) + f(n-2) & \text{if } n > 1 \end{cases} \tag{10}$$

5.2 More Examples about Recursion

More Examples

- **Tower of Hanoi**

  There are 3 pegs (A, B and C) and a tower of n disks on the first peg A, with the smallest disk on the top and the biggest at the bottom. The purpose of the puzzle is to move the whole tower from peg A to peg B, with the following simple rules:
  - Only one disk (the one at the top) can be moved at a time.
  - A bigger disk must not rest on a smaller disk.

    1. move top four disks from peg A to peg C
    2. move disk #5 from peg A to peg B
    3. move top four disks from peg C to peg B

    3.1 move top three disks from peg C to peg A
    3.2 move disk #4 from peg C to peg B
    3.3 move top three disks from peg A to peg B

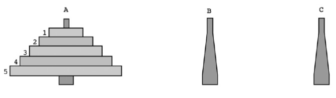


Figure 14: Initial

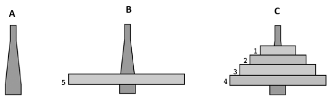


Figure 15: After Step 2

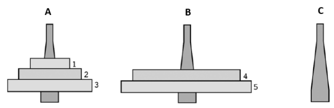


Figure 16: After Step 3.2

$\therefore$ To move n disks from `source` peg to `dest` peg using `temp` peg, if $n > 0$,
1. move $n-1$ disks from `source` peg to the `temp` peg using the `dest` peg
2. move disk $\#n$ from the `source` peg to the `dest` peg
3. move $n-1$ disks from `temp` peg to the `dest` peg using the `source` peg

```python
def towerHanoi(source, dest, temp, n):
    if n > 0:
        towerHanoi(source, temp, dest, n - 1)
        print('Move disk %d from peg %s to peg %s' % (n, source,
          dest))
        towerHanoi(temp, dest, source, n - 1)
```

```
>>> towerHanoi('A', 'B', 'C', 3)
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 3 from peg A to peg B
Move disk 1 from peg C to peg A
Move disk 2 from peg C to peg B
Move disk 1 from peg A to peg B
```

* *Every recursive call made, the 4^{th} parameter reduces, moving towards the terminal condition*
** *This is a void function – no* **`return`** *statement; if* `n == 0`*, function does nothing and exits*

- **Reversing a String**
  - defined by

$$
\begin{aligned}
('')^R &= '' \\
('a_1 a_2 \dots a_n')^R &= 'a_n \dots a_2 a_1' \\
&= ('a_n \dots a_2') + 'a_1' \\
&= ('a_2 \dots a_n')^R + 'a_1'
\end{aligned}
\tag{11}
$$

$$
\implies (s)^R = \begin{cases} s & \text{if } |s| = 0 \\ ('a_2 \dots a_n')^R + 'a_1' & \text{if } s = 'a_1 a_2 \dots a_n' \end{cases}
\tag{12}
$$

```python
def reverseStringR(s):
    if not s:
        return ''
    return reverseStringR(s[1:]) + s[0]
```

```
>>> reverseStringR('abcde')
'edcba'
```

- **Summing Up All Digits in an Integer**
  - defined by

$$
\begin{aligned}
sumDigR(20934) &= 2+0+3+9+4 \\
&= (2+0+3+9)+4 \\
&= sumDigR(2093)+4 \\
sumDigR(2) &= 2
\end{aligned}
\tag{13}
$$

$$
\implies sumDigR(n) = \begin{cases} sumDigR(\lfloor n/10 \rfloor) + (m \bmod 10) & \text{if } n > 9 \\ n & \text{if } n \leq 9 \end{cases}
\tag{14}
$$

```python
def sumDigR(n):
    ''' Works on non-negative integers '''
    if n > 9:
        newnumber = n // 10
        remainder = n % 10
        return sumDigR(newnumber) + remainder
    return n
```

```
>>> sumDigR(1234)
10
>>> sumDigR(4010010040)
10
>>> sumDigR(4)
4
```

- **Taylor Series – Sine**
  - defined by

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{(2n+1)} \quad = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \qquad \text{for all } x \tag{15}$$

  - iterative

$$\sin x = \sum_{n=0}^{k} \frac{(-1)^n}{(2n+1)!} x^{(2n+1)} \quad = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \qquad \text{for all } x \tag{16}$$

```python
def sinI(x, k):
    result = 0
    for n in range(0, k + 1):
        result += ((-1) ** n / fact(2 * n + 1) * x ** (2 * n + 1))
    return result
```

```
>>> sinI(pi / 6, 6)
0.5
```

  - recursive

$$\begin{aligned} \sin(x) \approxeq sinR(x,k) &= \sum_{n=0}^{k} \frac{(-1)^n}{(2n+1)!} x^{(2n+1)} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \\ sinR(x,k) &= \sum_{n=0}^{k-1} \frac{(-1)^n}{(2n+1)!} x^{(2n+1)} + \frac{(-1)^k}{(2k+1)!} x^{(2k+1)} \qquad \text{if } k \geq 0 \\ &= sinR(x,k-1) + \frac{(-1)^k}{(2k+1)!} x^{(2k+1)} \qquad \text{if } k \geq 0 \\ sinR(x,-1) &= \sum_{n=0}^{-1} \frac{(-1)^n}{(2n+1)!} x^{(2n+1)} = 0 \qquad \text{if } k < 0 \end{aligned} \tag{17}$$

$$\implies sinR(x,k) = \begin{cases} 0, & k < 0 \\ sinR(x,k-1) + frac(-1)^k (2k+1)! x^{(2k+1)}, & k \geq 0 \end{cases} \tag{18}$$

```python
def sinR(x, k):
    if k >= 0:
        return sinR(x, k - 1) + ((-1) ** k / fact(2 * k + 1) * x **
            (2 * k + 1))
    return 0
```

```
>>> sinR(pi / 6.6)
0.5
```

5.3 Variable Scope

Local Variable – variables created inside a function

- it only exists inside the function
  - ↪ *created* when the function is called
  - ↪ *deleted* when the call exits

↪ every function call creates a *new frame* to contain a *new set* of local variables

- parameters in the function definition behave like local variables
  - ↪ contain values passed into the function

Global Variable – variable defined in the main body of a file or the console

- visible
  - throughout the file or console
  - inside any file which imports that file
- `global` keyword – allows variable to be accessed from within a function call
  - does not create a local variable

  e.g.

```python
x = 0
def printx():
    global x
    print(x)
    x = x + 1
    print(x)
```

```
>>> printx()
0
1
>>> print(x)
1
```

  - bad practice! especially when modified in function scope
  - usually used as CONSTANTS[7]

    e.g.

    ```python
    POUNDS_IN_ONE_KG = 2.20462
    ```

Scope – extent of an area that a variable is considered relevant

- relevance of a local variable

  ↪ from the statement in which it is created → end of the function invocation
  - types of variable relevancy
    - **Using** a variable

      ↪ USE/access/read/referenced
    - **Defining** a variable

      ↪ DEF/define/update/assign

  e.g.

```python
import random
def printx(n):
    for i in range(0, n):
        x = random.randint(101, 303)
        print(x)
```

  - calling a *local* `x`

```
>>> printx(2)
263
268
>>> x
NameError: name 'x' is not defined
```

  - calling a *global* `x`

```
>>> x = 10
>>> printx(2)
299
166
>>> x
10
```

  * *the scope of a* `x` *is within the definition of* `printx`

  ↪ local variable of function `printx`; "lives" in `printx`; not accessible outside `printx`

  ** *Go up and go out but cannot go in!*
- Local v.s. Global variable

  e.g.

```
>>> x = 17
>>> def printx():
        print(x)
        print(x + 1)
>>> printx()
17
18
>>> x
17
```

```
>>> x = 83
>>> def printx():
        print(x)
        x = x + 1
        print(x)
>>> printx()
UnboundLocalError: local variable 'x'
↪   referenced before assignment
```

  - variable calls reference the variable assigned locally instead of the variable outside the function

[7] Convention is for variable names to be fully capitalized

5.4 Function Activations

Call stack – snapshot of the program execution

– shows which functions have been called but not yet completed the execution

e.g.

```
from math import sqrt
def hypotenuse(a, b):
    print('Into hypotenuse with a = ' + str(a) + ' and b = ' + str(b))
    res = sqrt(sum_of_squares(a, b))
    print('Out of hypotenuse with value ' + str(res))
    return res

def sum_of_squares(x, y):
    print('Into sum_of_squares with x = ' + str(x) + ' and y = ' + str(y))
    res = square(x) + square(y)
    print('Out of sum_of_squares with value ' + str(res))
    return res

def square(x):
    print('Into square with x = ' = str(x))
    res = x * x
    print('Out of square with value ' + str(res))
    return res
```

```
>>> hypotenuse(3, 4)
Into hypotenuse with a = 3 and b = 4
Into sum_of_squares with x = 3 and y
↪  = 4
Into square with x = 3
Out of square with value 9
Into square with x = 4
Out of square with value 15
Out of sum_of_squares with value 25
Out of hypotenuse with value 5.0
5.0
```

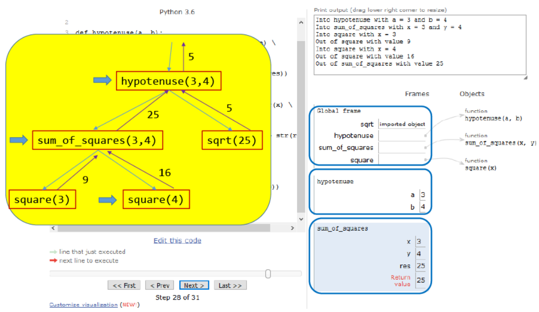


Figure 17: Examples of Call Tree & Call stack visualization (*pythontutor.com*)

5.5 Exercises

1. Which of the properties describes properly every time when executing reaches the beginning of the loop body?

```
def asc(lst):
    i = 0
    while i < len(lst):
        ### property ###
        if lst[i] > lst[i + 1]:
            return False
        i += 1
    return True
```

$\forall k:$ if $0 \leq k < i$ then $lst[k] \leq lst[k+1]$

$\forall k:$ if $i \leq k <$ `len(lst)` then $lst[k] \leq lst[k+1]$

$\forall k:$ if $0 \leq k <$ `len(lst)` then $lst[k] \leq lst[k+1]$

None of the above

* *Last option is correct for when the input* `lst` *is an null/empty list or only*

2. Given `lst = [1, 2, 3]`, what is the result of executing `asc(lst)`?

```
def asc(lst):
    i = 0
    while i < len(lst):
        ### property ###
        if lst[i] > lst[i + 1]:
            return False
        i += 1
    return True
```

- True
- False
- execution crashes
- Execution enters infinite loop
- None of the above

3. Suppose `n` can take any integer, when will `factorial(n)` produce different answer as `factorR(n)`?

```
def factorial(n):
    ans = 1
    i = 1
    while i <= n:
        ans = ans * i
        i = i + 1
    return ans

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

- ∞
- 0
- -5
- None of the above

4. `tryhere(5)` returns . . . ?

```
def tryhere(n):
    if n <= 0:
        return 1
    return n + tryhere(n - 2)
```

- 8
- 10
- 12
- Recursion Error

5. `tryThis(5)` returns . . . ?

```
def tryThis(n):
    if n == 0:
        return 1
    return n + tryhere(n - 2)
```

- 8
- 10
- 12
- Recursion Error

T1. What is printed by calling `qn1()`?

```
def qn1():
    x = 0
    def foo_printx():
        print(x)

    foo_printx()
    print(x)
qn1()
```

- 0 and 0
- 0 and 999
- 999 and 0
- 999 and 999
- None of the above

T2. What is printed by calling `qn2()`?

```python
def qn2():
    x = 0
    y = 999
    def foo_printx(y):
        print(y)

    foo_printx(x)
    print(x)
qn2()
```

0 and 0
0 and 999
system clash

T3. What is printed by calling `qn3()`?

```python
def qn2():
    x = 0
    def foo_printx():
        x = 999
        print(x)

    foo_printx(x)
    print(x)
qn3()
```

0 and 0
0 and 999
999 and 0
999 and 999
None of the above

5.6 Tutorial: Recursion

Boolean Logic – Short-Circuit Logic

- Logical Operators
  - `and` – once the result is `False`, adding more `True` to the right of `and` will never make it `True`
    ↪ once the preceding part evaluates to `False`, we do not evaluate the rest
    e.g.

```python
def foo():
    print("foo")
    return True

def bar(x):
    if x > 0 and foo():
        return "yes"
    return "no"
>>> print(bar(0))
no
```

```python
def foo(n):
    if n > 0 and 1 / n > 0.01:
        return 2 ** n
    return 0
>>> foo(0)
0
```

  - `or` – once the result is `True`, adding more `False` to the right of `or` will never make it `False`
    ↪ once the preceding part evaluates to `True`, we do not evaluate the rest
    e.g.

```python
def foo():
    print("foo")
    return True

def bar(x):
    if x > 0 or foo():
        return "yes"
    return "no"
```

```python
>>> print(bar(0))
foo
yes
>>> print(bar(1))
yes
```

Crossing Boundaries – modifying variables right outside scope

- `nonlocal`
  - indicates that the variable is not local
  - does not create a local variable 'x'

  e.g.

```
def qn_():
    x = 0
    def foo_printx():
        nonlocal x
        x = 999
        print(x)

    foo_print(x)
    print(x)
```

```
>>> qn_()
999
999
```

6 Lecture 06 (Tuples and Dictionaries)

6.1 Tuples

Tuples

- enclosed in parentheses
- separated by commas

  i.e.

  `(..., ..., ..., ...)`
- immutable – cannot be modified

  e.g.

```
>>> a_tuple = (12, 13, 'dog')
>>> a_tuple[1]
13
>>> a_tuple[1] = 9
TypeError: 'tuple' object does not
↪   support item assignment
```

```
>>> t1 = (1, 2, 3)
>>> t1.append(3)
AttributeError: 'tuple' object has no
↪   attribute 'append'
>>> t1.remove(1)
AttributeError: 'tuple' object has no
↪   attribute 'remove'
```

- passed by value to a function

  ↪ write protected

Lists

- homogeneous – stores a large collection of data of the same type

  e.g.

  list of 200 student records in a class

Tuples

- heterogeneous – stores a small collection of items of multiple data types/concepts

  e.g.

  a single student record with name (`str`), student number (`str`) & marks (`int`)

Tuple Assignment

e.g.

```
>>> a = 50
>>> b = 99
>>> a, b
(50, 99)
```

```
>>> temp = a
>>> a = b
>>> b = temp
>>> a, b
(99, 50)
```

```
>>> a, b = b, a
>>> a, b
(99, 50)
```

```
>>> a, b = a - b, b - a
>>> a, b
(49, -49)
```

Tuple as Return Values

- `divmod(..., ...)` returns 2 values – quotient & remainder, as a tuple

  e.g.

  ```
  >>> t = divmod(41, 5)
  >>> t
  (8, 1)
  ```

  ```
  >>> quotient, remainder = divmod(41, 5)
  >>> quotient, remainder
  (8, 1)
  ```

e.g.

```
def sum_ave(lst):
    total = sum(lst)
    n = len(lst)
    return total, total / n
```

```
>>> lst = [1, 2, 3, 4, 5, 6]
>>> total, average = sum_ave(lst)
>>> total, average
(21, 3.5)
```

Assigning only 1 element to a Tuple

- a comma must follow the only element in a tuple with one element

  i.e.

  ```
  >>> c_tuple = (3, )
  >>> print(c_tuple)
  (3, )
  >>> type(c_tuple)
  <class 'tuple'>
  >>> c_tuple[0]
  3
  ```

* *without a comma, Python will treat the parentheses as an arithmetic operation*

Tuple Operations

- "**Changing Content**" – creates a new tuple

  e.g.

  ```
  >>> tup1 = (1, 2, 3)
  >>> tup2 = (4, 5, 6)
  >>> tup3 = tup1[0:2] + tup2
  >>> tup3
  (1, 2, 4, 5, 6)
  >>> tup1
  (1, 2, 3)
  ```

Examples of Uses

e.g.

- Recording the locations of 100 nice restaurants in Singapore as the coordinate values of x & y

  e.g.

  ```
  locations_of_nice_restaurants = [(100, 50), (30, 90), (50, 90)]
  ```

6.2 Dictionaries

Python Dictionary[8]

i.e.

```
{key1: value1, key2: value2}
```

- mutable collection of key-value pair – each key has a corresponding value

  i.e.

  <key>: *<value>*

  - *<key>* maps to *<value>*

  * *<key> on the left; <value> on the right*
- can store any type
- searches for the *key* in the dictionary; dictionary look-up using key

  ↪ system looks up for its *value*

  e.g.

  ```
  >>> students = {'A100000X': 'John', 'A123456X': 'Peter', 'A999999X': 'Paul'}
  >>> students['A123456X']
  'Peter'
  >>> my_dictionary = {'a': 1, 'b': 2}
  >>> my_dictionary['b']
  2
  ```
- dictionaries can be initialized as empty

  e.g.

  ```
  >>> da = {}
  >>> type(da)
  <class 'dict'>
  ```

Hashable

- **Hash** – function that takes a value (of any kind) and returns an integer
- dictionaries uses hash values to store and look up key-value pairs
- keys in a dictionary has to be hashable which has to be **immutable**

  ↪ tuples can be keys and values in a dictionary

  e.g.

  ```
  >>> tup = (1, 2, 3)
  >>> dict[(1, 2, 3)] = 'wow'
  >>> dict
  {(1, 2, 3): 'wow'}
  ```

  ↪ list can be values in a dictionary; **but not keys**

  e.g.

  ```
  >>> lst = [1, 2, 3]
  >>> dict = { }
  >>> dict[lst] = 'oops'
  TypeErrorL unhashable type: 'list'
  ```

[8] Called Hash Table in some other languages

Examples of Uses

- storing locations of restaurants (recorded as the coordinate value of x and y) and names

  e.g.

```
>>> locations = {(10, 30): 'MacDonald', (30, 99): 'Burger King', (22, 33):
↪ 'Pizza Hut'}
>>> locations[(22, 33)]
'Pizza Hut'
```

- tracking stocks of fruits

  e.g.

```
>>> my_stock = {"apples": 450,
↪ "oranges": 412}
>>> my_stock["apples"]
450
>>> my_stock["apples"] +
↪ my_stock["oranges"]
862
```

- get an associated operation

  e.g.

```
>>> my_alphabet_index = {'a': 1,
↪ 'b': 2, ..., 'z': 26}
>>> my_alphabet_index['z']
26
```

Dictionary manipulation

- **Accessing Key-value Pairs**[9]
  - very fast! – almost instantaneous
  - ways to access
    - `<dict>[<key>]` – "key indexing"

      ↪ will crash if the key does not exist
    - `<dict>.get(<value>, default = None)`

      ↪ will not crash if the key does not exist
      - returns `<dict>[<key>]` if exists; default value otherwise

  e.g.

```
>>> my_fruit_inventory = {"apples": 450, "oranges": 200}
>>> my_fruit_inventory["apples"]
450
>>> my_fruit_inventory.get("apples")
450
>>> my_fruit_inventory["pears"]
KeyError!
>>> my_fruit_inventory.get("pears")
None
```

- **Adding/Updating Key-value Pairs**
  - cannot have duplicate keys in a dictionary

    ↪ assigning a value to an existing key will replace the existing value

  e.g.

```
>>> my_fruit_inventory = {"apples": 450, "oranges": 200}
>>> my_fruit_inventory["pears"] = 100
>>> print(my_fruit_inventory)
{"apples": 450, "oranges": 200, "pears": 100}
>>> my_fruit_inventory["oranges"] = 100
>>> print(my_fruit_inventory)
{"apples": 450, "oranges": 100, "pears": 100}
>>> my_fruit_inventory["orange"] += 100
>>> print(my_fruit_inventory)
{"apples": 450, "oranges": 200, "pears": 100}
```

[9] constant time

- **Deleting Key-value Pairs**

  e.g.

```
>>> my_fruit_inventory = {"apples": 450, "oranges": 200}
```

```
>>> my_fruit_inventory.pop("apples")
450
>>> print(my_fruit_inventory)
{'oranges': 200}
```

```
>>> del my_fruit_inventory["apples"]
>>> print(my_fruit_inventory)
{'oranges': 200}
```

  * *deleting non-existent keys will return* `KeyError`

- **Other Dictionary Methods**
  - `<dict>.clear()` – removes all entries
  - `del <dict>` – deletes the dictionary
  - `<dict>.copy()` – makes a copy
  - `<dict>.keys()` – returns all keys
  - `<dict>.values()` – returns all values
  - `<dict>.items()` – returns all keys & values

Examples

- **Frequency Count**

  Given a character string, keep the frequency of occurrences of each character.

```python
def frequency(aString):
    freqD = { }
    for char in aString:
        if char in freqD:
            freqD[char] += 1
        else:
            freqD[char] = 1
    return freqD
```

```
>>> fd = frequency('incomprehensibilities')
>>> fd
{'i': 5, 'n': 2, 'c': 1, 'o': 1, 'm': 1, 'p': 1, 'r': 1, 'e': 3, 'h': 1, 's':
↪  2, 'b': 1, 'l': 1, 't': 1}
```

  - *Examples on using other dictionary methods*
    - `<dict>.keys()`

      e.g.

```
>>> fd.keys()
dict_keys(['i', 'n', 'c', 'o', 'm', 'p', 'r', 'e', 'h', 's', 'b', 'l',
↪  't'])
>>> type(fd.keys())
<class 'dict_keys'>
```

    - `<dict>.values()`

      e.g.

```
>>> fd.values()
dict_values([5, 2, 1, 1, 1, 1, 1, 3, 1, 2, 1, 1, 1])
```

    - `<dict>.items()`

      e.g.

```
>>> fd.items()
dict_items([('i', 5), ('n', 2), ('c', 1), ('o', 1), ('m', 1), ('p',
↪  1), ('r', 1), ('e', 3), ('h', 1), ('s', 2), ('b', 1), ('l', 1),
↪  ('t', 1)])
```

– **Iterating over a Dictionary**

```python
def prt1_d(dict):
    print('Iterate over dictionary')
    for key in dict:
        print('{} -> {}'.format(key, dict[key]))
```

```python
def prtK_d(dict):
    print('Iterate over dictionary keys')
    for key in dict.keys():
        print('{} -> {}'.format(key, dict[key]))
```

```python
def prtI_d(dict):
    print('Iterate over dictionary items')
    for key in dict.keys():
        print('{} -> {}'.format(key, val))
```

```
>>> prt1_d(fd)
Iterate over dictionary
i -> 5
n -> 2
c -> 1
o -> 1
m -> 1
p -> 1
r -> 1
e -> 3
h -> 1
s -> 2
b -> 1
l -> 1
t -> 1
```

```
>>> prtK_d(fd)
Iterate over dictionary
↪ keys
i -> 5
n -> 2
c -> 1
o -> 1
m -> 1
p -> 1
r -> 1
e -> 3
h -> 1
s -> 2
b -> 1
l -> 1
t -> 1
```

```
>>> prtI_d(fd)
Iterate over dictionary
↪ items
i -> 5
n -> 2
c -> 1
o -> 1
m -> 1
p -> 1
r -> 1
e -> 3
h -> 1
s -> 2
b -> 1
l -> 1
t -> 1
```

– **Reverse Lookup – Iterating over Dictionary Values**

Given a value `v`, return the first key that maps to `v`

- problems
  1. there may be *more than one key* that maps to `v`
  2. no simple syntax to do a reverse lookup
     ↪ search while looking up

```python
def reverse_lookup(dict, val):
    for key in dict:
        if dict[key] == val:
            return key
    raise LookupError()
```

```
>>> reverse_lookup(fd, 2)
'n'
>>> reverse_lookup(fd, 3)
'e'
>>> reverse_lookup(fd, 4)
LookupError
```

- **Inverting a Dictionary**
- **Fibonacci Function in Memoized Form**

```
def fibM(n):
    memo = {}
    def memofib(n):
        if n <= 1:
            return 1
        if n in memo:
            return memo[n]
        res = memofib(n - 1) + memofib(n - 2)
        memo[n] = res
        return res

    return memofib(n)
```

```
>>> fibM(15)
987
>>> fibM(35)
14930352
>>> fibM(50)
20365011074
```

6.3 Tutorial: Tuples, Dictionaries

Tuple Operations

- len(*<tuple>*) – returns the number of elements
- *<element>* in *<tuple>*
  - returns True if *<element>* is in *<tuple>*, and False otherwise
- for *<var>* in *<tuple>*
  - iterates over all the elements of *<tuple>*; each element stored in *<var>*
  - max(*<tuple>*) – returns the maximum element
  - min(*<tuple>*) – returns the minimum element

Tuple Access

- use square brackets [...] to retrieve an element in a tuple
- 2 types of index for a tuple of size n
  - Forward index – starts from 0 ends at n - 1
  - Backward index – starts from -1 ends at -n

e.g.

```
>>> tup = (1, 2, 3, (4, 5), 6, 7)
>>> tup[2]
3
>>> tup[-2]
6
>>> tup[3]
(4, 5)
```

0	1	2	3	4	5	6	7
e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
-8	-7	-6	-5	-4	-3	-2	-1

Range Type

- sequence of numbers
- immutable

Dictionary

- Creating a dictionary from a tuple/list
  i.e.
  ```
  dict(sequence_of_pairs)
  ```
  - the element in the tuple/list must be a pairs
  - the first of the pair will be the key; the second of the pair will be the value
    i.e.
    `sequence_of_pairs = [(<key>, <value>), ...]`
- **Dictionary Operations**
  - `<key> in <dict>`
    - returns `True` if `<element>` is in `<tuple>`, and `False` otherwise
  - `len(<dict>)` – returns the number of elements

7 Lecture 07 (Higher-Order Functions, Maps, Filters)

7.1 Higher-Order Functions

First-order Function

- group of statements that exist within a program for the purpose of performing a specific task
- can be defined; fixed name
  e.g.
  ```python
  def compare(lst, num):
      if not lst:
          return 0
      else:
          cnt = compare(lst[1:], num)
          if lst[0] < num:
              return cnt + 1
          else:
              return cnt
  ```
- support reuse – supply of different arguments to the function
  e.g.
  ```
  >>> compare([1, 2, 3, 4], 2.5)
  2
  >>> compare('animals', 'f')
  2
  ```
- *callable* – a variable that is a function
  - can only use functions by applying appropriate arguments to it
  e.g.
  ```
  >>> callable(lt)
  True
  >>> callable(compare)
  True
  ```
  ```
  >>> x = 3
  >>> callable(x)
  False
  ```
- helper functions defined locally
  e.g.
  ```python
  def distance(x1, y1, x2, y2):
      def square(x):
          return x * x

      return sqrt(square(x1 - x2) + square(y1 - y2))
  ```
  ```
  >>> distance(1, 2, 3, 4)
  2.8284271247461903
  >>> square(3)
  NameError: name 'square' is not defined
  ```

Higher-order Function

- function that takes functions as arguments or returns a function

  e.g.

```python
def compareG(lst, num, op):
    if not lst:
        return 0
    else:
        cnt = compareG(lst[1:], num, op)
        if op(lst[0], num):
            return cnt + 1
        else:
            return cnt
```

- supports *even more* code reuse

  e.g.

```python
def lt(a, b):
    return a < b

def geq(a, b):
    return a >= b

def modulo(a, b):
    return a % b == 0

def faraway(a, b):
    return a >= b ** 2
```

```python
>>> compareG([1, 2, 3, 4], 2.5, lt)
2
>>> compareG('animals', 'f', geq)
5
>>> compareG([23, 17, 40, 104], 8, modulo)
2
>>> compareG([23, 17, 40, 104], 5, faraway)
2
```

- assignment to a variable[10]

  e.g.

```python
>>> x = lt
>>> y = x
>>> z = y
>>> compareG([1, 2, 3, 4], 2.5, z)
2
```

- **Returned from a Function**

  e.g.

```python
def adjust(x):
    def double(price):
        return price * 2

    def half(price):
        return price / 2

    if x == 'Y':
        return double
    else:
        return half
```

```python
def setPrice(lst, res):
    if not lst:
        return res
    else:
        x, y = lst[0]
        action = action(x)
        res.append(action(y))
        return setPrice(lst[1:], res)
```

```python
>>> products = [('Y', 40), ('N', 50), ('Y', 30), ('N', 40)]
>>> setPrice(products, [])
[80, 25.0, 60, 20.0]
```

[10] Functions can be passed as arguments to another function and be stored in a local variable/parameter

- **Stored in a List**

  e.g.

```python
def pipe(lst, num)
    if not lst:
        return num
    else:
        num2 = (lst[0])(num)
        return pipe(lst[1:], num2)
```

```python
def double(x):
    return x + x
```

```
>>> pipe([double, double, double], 3)
24
```

- **Remain Anonymous in an Expression**
  - not bound by an explicit identifier; need not have a name

  e.g.

```
>>> lambda x : x ** 4
<function <lambda> at 0x04926DB0>
>>> (lambda x : x ** 4)(4)
256
```

```
>>> fvar = lambda x : x ** 4
>>> fvar
<function <lambda> at 0x04926DB0>
>>> fvar(4)
256
```

Lambda Expressions

i.e.

`lambda <arguments> : <expr>`

- `lambda` – keyword
- `<arguments>` – parameters
- `<expr>` – body

e.g.

```python
def add1(x):
    return x + 1
```

```
>>> add1(9)
10
```

```python
func = lambda x : x + 1
```

```
>>> func(9)
10
```

- does not need a '`return`' statements
- expression; not a sequence of statements
- "handy" – can be defined and used whenever you want

  e.g.

```
>>> compareG([1, 2, 3, 4], 2.5, lambda x, y : x<y)
2
>>> compareG('animals', 'f', lambda x, y : x >= y)
5
>>> compareG([23, 17, 40, 104], 8, lambda a, b : a % b == 0)
2
>>> compareG([23, 17, 40, 104], 5, lambda a, b : a >= b ** 2)
2
```

Examples

- **Composite Function**

```python
def compose(f, g):
    return lambda x : g(f(x))
```

$$(f \circ g)(x) = g(f(x)) \tag{19}$$

```
>>> qd = compose(double, double)
>>> qd(4)
16
>>> qd('abd')
'abdabdabdabd'
```

- **Approximated Derivative**

  Given a function f, the derivative of f is

$$\frac{d\ f(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{20}$$

  which can be approximated by choosing a very small number dx

$$\frac{d\ f(x)}{dx} \approx \frac{f(x+dx) - f(x)}{dx} \tag{21}$$

```
def deriv(f):
    dx = 0.000000001
    return lambda x : (f(x + dx) - f(x)) / dx
```

  e.g.

$$\frac{d \sin x}{dx} = \cos x \tag{22}$$

```
>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
```

$$\frac{d\big(x^3 + 3x - 1\big)}{dx} = 3x^2 + 3 \tag{23}$$

```
def f(x):
    return x ** 3 + 3 * x - 1
```

```
>>> deriv(f)(9)
246.00001324870388
>>> x = 9
>>> 3 * x ** 2 + 3
246
```

- **Simplified Newton's Method**

  To compute root of function $g(x)$

  1. Estimate an answer x
  2. If $g(x) \approx 0$, x is a root
  3. Else, find x_1 such that

$$f'(x) = \frac{g(x)}{x - x_1} \tag{24}$$

  4. Repeat Step 2 with $x = x_1$

```
def newtonM(g):
    x = 999 ## doesn't matter
    err = 0.0000000001
    while(abs(g(x)) > err):
        x = x - g(x) / deriv(g)(x)
    return x
```

  e.g.

  - *Finding the Square Root of a number*

    ↪ equivalent to solving the equation: $x^2 - A = 0$

      ↪ equivalent to finding the root of the function: $x^2 - A$

    e.g.

```
def my_own_sqrt(A):
    return newtonM(lambda x : x * x - A)
```

```
>>> x = my_own_sqrt(10)
>>> x * x
9.999999999999998
```

- *Computing* $\log_2(N)$
  ↪ equivalent to solving the equation: $2^x - N = 0$
    ↪ equivalent to finding the root of the function: $2^x - N$

  e.g.

```
## solving the log: 2 ** x - N = 0
def my_own_log2(n):
    return newtonM(lambda x : 2 **
    ↪  x - n)
```

```
>>> x = my_own_log2(234)
7.870364719583405
>>> 2 ** 7.870364719583405
234.00000000000003
>>> my_own_log2(100)
6.643856189774724
>>> 2 ** 6.643856189774724
99.99999999999997
```

7.2 Map and Filter, and going deeper

Our `map()`

- can only process lists;
- cannot work on other sequences like tuples, strings, etc.

```
def map(f, seq):
    output = []
    for i in seq:
        output.append(f(i))
    return output
```

e.g.

- *squaring a sequence*

$$f(i) = i \times i \tag{25}$$

```
square = lambda i : i * i
```

- *scaling a sequence*

$$f(i) = i \times n \tag{26}$$

```
scale2 = lambda i : i * 2
```

```
lst = [5, 1, 4, 9, 11, 22, 12, 55]
>>> map(square, lst)
[25, 1, 16, 81, 121, 484, 144, 3025]
>>> map(scale2, lst)
[10, 2, 8, 18, 22, 44, 24, 110]
>>> map(lambda x : x * x, lst)
[25, 1, 16, 81, 121, 484, 144, 3025]
>>> map(lambda x : 2 * x, lst)
[10, 2, 8, 18, 22, 44, 24, 110]
>>> map(lambda x : -x, lst)
[-5, -1, -4, -9, -11, -22, -12, -55]
```

Python's original version of `map()` – applies a function `f` to every element `x` in the sequence

- returns a type `map` object
  - ↪ can convert into other sequences like `list` or `tuple`

  e.g.
  ```
  >>> tup = (1, -2, 3)
  >>> map1 = map(abs, tup)
  >>> map1
  <map object at 0x112e61438>
  >>> type(map1)
  <class 'map'>
  ```
- `map` object is an iterable
  - ↪ can be converted to a `list`/`tuple`

    e.g.
    ```
    >>> map1List = tuple(map1)
    >>> map1List
    [1, 2, 3]
    >>> map1Tuple = list(map1)
    >>> map1Tuple
    ()
    ```
    ```
    >>> map1Tuple = tuple(map1)
    >>> map1Tuple
    (1, 2, 3)
    >>> map1List = list(map1)
    >>> map1List
    []
    ```
  - ↪ after items are "taken out" from the `map` object, the items will be "gone"[11]
- powerful tool in Python
  - ↪ allows you to perform a lot of operations with less redundant code

Python's `filter()` – applies a predicate[12] function `f` to every element `x` in the sequence

- returns an iterable
  - ↪ keeps the item `x` in the iterable if `f(x)` returns `True`;
  - ↪ removes the item `x` in the iterable if `f(x)` returns `False`

  e.g.
  ```
  >>> l = [1, 2, 3, 'a', (1, 2), ('b', 3)]
  >>> filter(lambda x : type(x) == int, l)
  <filter object at 0x112e618d0>
  >>> list(filter(lambda x : type(x) == int, l))
  [1, 2, 3]
  >>> l = [1, 2, 'a', (1, 2), 6, ('b', 3), 999]
  >>> list(filter(lambda x : type(x) == int, l))
  [1, 2, 6, 999]
  >>> list(filter(lambda x : type(x) == str, l))
  ['a']
  >>> l2 = [1, 4, 5, -4, 9, -99, 0, 32, -9]
  list(filter(lambda x : x < 0, l2))
  [-4, -99, -9]
  ```

[11] `map` objects can be converted to a `list` or `tuple` only once

[12] A function that return `True` or `False`

Deep Operations

- **Generalization** – `deepMap(...)`
  - takes in 2 arguments – a function and a sequence
  - applies function on every element in the sequence *and nested sequences*

```python
def deepMap(func, seq):
    if seq == []:
        return seq
    elif type(seq[0]) != list:
        return [func(seq[0])] + deepMap(func, seq[1:])
    else:
        return [deepMap(func, seq[0])] + deepMap(func, seq[1:])
```

```
>>> l = [1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
>>> deepMap(str, l)
['1', '2', '3', ['1', '2'], ['2', '3', '4', ['1', '2', '3']], ['3', '4', '5']]
>>> deepMap(lambda x : x / 2, l)
[0.5, 1.0, 1.5, [0.5, 1.0], [1.0, 1.5, 2.0, [0.5, 1.0, 1.5]], [1.5, 2.0, 2.5]]
```

  e.g.
  - *deep copy*

    i.e.

    ```
    >>> l = [1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
    >>> l2 = deepMap(lambda x : x, lst)
    >>> lst[3][0] = 999
    >>> lst
    [1, 2, 3, [999, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
    >>> l2
    [1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
    ```

- `flatten(...)`

  Given a nested list, output a list with all the elements but without any sublist structures

```python
def flatten(seq):
    if seq == []
        return seq
    elif type(seq[0]) != list:
        return [seq[0]] + flatten(seq[1:])
    else:
        return flatten(seq[0]) + flatten(seq[1:])
```

```
>>> l = [1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
>>> flatten(l)
[1, 2, 3, 1, 2, 2, 3, 4, 1, 2, 3, 3, 4, 5]
```

- useful to solve problems with non-linear data

  e.g.
  - trees
    - copying a directory – when the directory contains a lot of files in many subdirectories
    - computer animation – skeleton animation
    - shortest path tree
  - n-dim arrays
    - image processing
      - *an image is a list (rows) of lists (columns) of lists (RGB values)*
      - map a function to change certain values – changing colors
  - graphs

7.3 Exercises

1. Why does execution of the call `fibM(40)` still take a long time?

```
def fibM(n):
    memo = {0: 1, 1: 1}
    if n <= 1:
        return memo[n]
    else:
        fn_1 = fibM(n - 1)
        fn_2 = fibM(n - 2)
        memo[n] = fn_1 + fn_2
        return memo[n]

fibM(40)
```

Every call of the function `fibM` creates a new local variable `memo`. Calls to `memo` accesses the newly created `memo` in each previous `fibM` call which only has `{0: 1, 1: 1}`

* *accumulator variable should be declared outside the recursive function definition*

2. Given that `f` is defined as follows, which of the following calls to `f` are valid?

```
f = lambda a : lambda b : a + b
```

`f(2, 3)`
`f(2)(3)`
`f(2)` – returns a function – "partial function"

3. Given the following assignment to variable `g`, which of the following expressions return `1`?

```
g = lambda a, b : b if a > 0 else -b
```

`g(-2, -1)`
`g(3)(1)`
`g(4, 1)`

7.4 Tutorial: HOF, Map, Filter, Reduce

Lambda Expressions

e.g.

```
(lambda a : lambda b : lambda c : a + b + c)
```

- takes 1 argument, `a`
  ↪ returns a function which takes 1 argument, `b`
    ↪ returns a function which takes 1 argument, `c`
      ↪ returns the result `a + b + c`

  * *Takes in 3 arguments : Evaluates to a number*
- binding environment
- similar to:
  i.e.

```
def f(a):
    def g(b):
        def h(c):
            return a + b + c

        return h(c)
    return g(b)
```

- brackets following a functions tells Python to pass whatever is in the bracket into the function as arguments
- consecutive applications of functions is treated as left associative

any()

e.g.

- If `L = [1, 2, 3, 4]`, are there any numbers that is greater than 3 in L?
  e.g.
  ```
  >>> L = [1, 2, 3, 4]
  >>> any(x > 3 for x in L)
  True
  ```
- If `L = [1, 2, 3, 4]`, are there any numbers that is greater than 9 in L?
  e.g.
  ```
  >>> any(x > 9 for x in L)
  False
  ```
- Are there any prime numbers in the lists?
  e.g.
  ```
  >>> any(isPrime(x) for x in [4,
  ↪ 6, 8, 9, 99])
  False
  >>> any(isPrime(x) for x in [4,
  ↪ 6, 8, 9, 97, 99])
  True
  ```

all()

e.g.

- If `L = [1, 2, 3, 4]`, are all numbers in L greater than 3? (and 0?)
  e.g.
  ```
  >>> all(x > 3 for x in L)
  False
  >>> all(x > 0 for x in L)
  True
  ```
- Are all the numbers in the list prime numbers?
  e.g.
  ```
  >>> all(isPrime(x) for x in [4,
  ↪ 6, 8, 9, 99])
  False
  >>> all(isPrime(x) for x in [3,
  ↪ 5, 7, 11, 97])
  True
  ```

8 Lecture 08 (Multi-Dimensional Arrays)

8.1 Multi-Dimensional Arrays

Two Dimensional Lists – a list that has other lists as its elements[13]

e.g.

```
## a 3 x 5 table
table = [ [4, 2, 1, 0, 0],   ## row 0
          [8, 3, 3, 1, 6],   ## row 1
          [0, 0, 0, 0, 0] ] ## row 2
```

	0	1	2	3	4
0	4	2	1	0	0
1	8	3	3	1	6
2	0	0	0	0	0

```
>>> table[1]
[8, 3, 3, 1, 6]
>>> table[1][3]
1
```

```
def main():
    ## local function for printing specific array
    def prtTable():
        print('[')
        for i in range(3):
            print('[', end = ' ')
            for j in range(5):
                print(table[i][j], end = ' ')
            print(']')
        print(']')

    ## display the table
    prtTable()
    ## update the third row with the sum of first two rows
    for j in range(5):
        table[2][j] = table[0][j] + table[1][j]
```

[13] Accessing elements – left associativity

```
    ## display the table
    prtTable()
    ## update the last column by the average of the sum of each row
    for i in range(3):
        table[i][-1] = sum(table[i]) / len(table[i])
    prtTable()
```

```
>>> main()
[
[ 4 2 1 0 0 ]
[ 8 3 3 1 6 ]
[ 0 0 0 0 0 ]
]
[
[ 4 2 1 0 0 ]
[ 8 3 3 1 6 ]
[ 12 5 4 1 6 ]
]
[
[ 4 2 1 0 1.4 ]
[ 8 3 3 1 4.2 ]
[ 12 5 4 1 5.6 ]
]
```

Matrix Addition

i.e.

$$A + B = C$$
$$\begin{pmatrix} 10 & 21 & 7 & 9 \\ 4 & 6 & 14 & 5 \end{pmatrix} + \begin{pmatrix} 3 & 7 & 18 & 20 \\ 6 & 5 & 8 & 15 \end{pmatrix} = \begin{pmatrix} 12 & 28 & 25 & 29 \\ 10 & 11 & 22 & 20 \end{pmatrix} \tag{27}$$

```
## matrix addition
## mtxC = mtxA + mtxB
def madd(mtxA, mtxB):
    m = len(mtxA)           ## number of rows
    n = len(mtxA[0])        ## number of columns
    mtxC = []               ## ----------
    for i in range(m):      ## create a zero
        lst = []            ## matrix mtxC
        for j in range(n):  ## of dimension
            lst.append(0)   ## m x n
        mtxC.append(lst)    ## -----------
    ## matrix addition
    for i in range(m):
        for j in range(n):
            mtxC[i][j] = mtxA[i][j] + mtxB[i][j]
    return mtxC
```

↪ creates a zero matrix – `mtxC`
↪ “in-place” update of every element in `mtxC`

List Comprehension

i.e.

```
[ <expr> for <element> in <iterable> ]
```

e.g.

```python
lst = [ i for i in range(3) ]
```

```python
lst = []
for i in range(3):
    lst.append(i)
```

```python
lst2 = [ j for i in range(3)
         for j in range(5) ]
```

```python
lst2 = []
for i in range(3):
    for j in range(5):
        lst2.append(j)
```

Alternative Generation of Matrix Elements

```python
## create mtxC of zero matrix
mtxC = [
    [ 0 for j in range(n) ]
for i in range(m) ]
```

```python
matC = [ [0] * n for i in range(m) ]
```

Efficient Generation of Matrix Elements

– **Zero Matrix of $m \times n$**

```python
mat1 = [
    [ 0 for j in range(n) ]
for i in range(m) ]
```

```
>>> prtTable(mat1)
[
[ 0 0 0 0 0 ]
[ 0 0 0 0 0 ]
[ 0 0 0 0 0 ]
]
```

– **Random Numbered Matrix of $m \times n$**

```python
mat2 = [
    [ randint(0, 20) for j in range(n) ]
for i in range(m) ]
```

```
>>> prtTable(mat2)
[
[ 2 14 2 15 20 ]
[ 9 19 12 7 4 ]
[ 3 6 16 8 5 ]
]
```

– **Numbered Matrix of $m \times n$**

```python
mat3 = [
    [ i *  n + j for j in range(n) ]
for i in range(m) ]
```

```
>>> prtTable(mat3)
[
[ 0 1 2 3 4 ]
[ 5 6 7 8 9 ]
[ 10 11 12 13 14 ]
]
```

– **Identity (Square) Matrix**

```python
id = [
    [ 0 if i != j else 1 for j in range(n) ]
for i in range(n) ]
```

```
>>> prtTable(id)
[
[ 1 0 0 0 0 ]
[ 0 1 0 0 0 ]
[ 0 0 1 0 0 ]
[ 0 0 0 1 0 ]
[ 0 0 0 0 1 ]
]
```

Pretty Printing of Matrices

e.g.

```
>>> mtx = [
        [ i * 10 + j for j in range(10) ]
    for i in range(4) ]
>>> mtx
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11, 12, 13, 14, 15, 16, 17, 18, 19], [20,
↪  21, 22, 23, 24, 25, 26, 27, 28, 29], [30, 31, 32, 33, 34, 35, 36, 37, 38,
↪  39]]
>>> from pprint import pprint
>>> pprint(mtx)
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39]]
```

More Examples – List Comprehension

– **Generating an Odd-numbered List**

```
lst1 = [ 2 * i + 1 for i in range(20) ]
```

```
>>> lst1
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39]
```

– **Generating a List in Reverse**

```
def rlst(n):
    return [ n - i - 1 for i in range(n) ]
```

```
>>> rlst(20)
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

– **Generating a List of Leap-years**

```
## leap-year list from 1900 to 2050
## it is divisible by 4, and it should not be divisible by 100 unless it
↪  is also divisible by 400
leap_years = [ y for y in range(1900, 2051, 4) if y % 100 != 0 or (y %
↪  100 == 0 and y % 400 == 0) ]
```

```
>>> leap_years
[1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936, 1940, 1944, 1948, 1952,
↪  1956, 1960, 1964, 1968, 1972, 1976, 1980, 1984, 1988, 1992, 1996, 2000, 2004,
↪  2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036, 2040, 2044, 2048]
```

– **Generating a List of Nonprime and Prime Numbers**

```
def nonprime(n):
    root = sqrt(n)
    if root == int(root):
        limit = int(root)
    else:
        limit = int(root) + 1
    return [ j for i in range(2, limit)
        for j in range(i * 2, n, i) ]

def prime(n):
    notprimes = nonprime(n)
    return [ x for x in range(2, n) if x not in notprimes ]
```

```
>>> nonprime(50)
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44,
↪ 46, 48, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 8, 12, 16,
↪ 20, 24, 28, 32, 36, 40, 44, 48, 10, 15, 20, 25, 30, 35, 40, 45, 12, 18, 24,
↪ 30, 36, 42, 48, 14, 21, 28, 35, 42, 49]
>>> prime(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Sub-matrix of a Matrix

e.g.

```
mat = createM(5, 5, 99)
print('mat is ...')
prtTable(mat)
```

```
mat is ...
[
[ 39 82 25 24 79 ]
[ 86 79 54 32 89 ]
[ 28 26 84 89 35 ]
[ 12 58 81 17 70 ]
[ 95 55 94 89 19 ]
]
```

– indexing/slicing
  – `mat[2][4]` is equivalent to
    i.e.
    ```
    >>> row2 = mat[2]
    >>> row2[4]
    ```
  – `mat[2:4][:]` is equivalent to
    i.e.
    ```
    >>> row23 = mat[2:4]
    >>> row23[:]
    ```
    e.g.
    ```
    sub_rows = mat[2:4]
    print('sub_rows is
    ↪ ...')
    prtTable(sub_rows)
    ```
    ```
    sub_rows is ...
    [
    [ 28 26 84 89 35 ]
    [ 12 58 81 17 70 ]
    ]
    ```
  – `mat[:][2:4]` is equivalent to
    i.e.
    ```
    >>> what = mat[:]
    >>> what[2:4]
    ```
    e.g.
    ```
    sub_cols =
    ↪ mat[::][2:4]
    print('sub_cols is
    ↪ ...')
    prtTable(sub_cols)
    ```
    ```
    sub_cols is ...
    [
    [ 28 26 84 89 35 ]
    [ 12 58 81 17 70 ]
    ]
    ```

- **Get Sub-matrix of a Matrix**

```
def subMat(mat, r1, r2, c1, c2):
    m = r2 - r1 + 1
    n = c2 - c1 + 1
    ans = createM(m, n, 1)
    for i in range(m):
        for j in range(n):
            ans[i][j] = mat[r1 + i][c1 + j]
    return ans

sub_mats = subMat(mat, 2, 4, 2, 4)
print('sub_mats is ...')
prtTable(sub_mats)
```

```
sub_mats is ...
[
[ 84 89 35 ]
[ 81 17 70 ]
[ 94 89 19 ]
]
```

Application – Smudge the Paint
Given the position of a cell, if the cell is `oldColor`, turn it and all neighboring (including diagonal) cells which are `oldColor` into `newColor`.

- iteratively

```
def smudge(paint, i, j, oldColor, newColor):
    ## local painting function
    def neighbor(x, y, wlst, vlst):
        news = []
        for i in [x - 1, x, x + 1]:
            if i < 0 or i >= len(paint):
                continue
            for j in [y - 1, y, y + 1]:
                if j < 0 or j >= len(paint[0]):
                    continue
                if paint[i][j] != oldColor:
                    continue
                if (i, j) in vlst:
                    continue
                if (i, j) in wlst:
                    continue
                news.append((i, j))
        return news

    def touch(worklist, visited):
        while worklist:
            i, j = worklist[0]
            paint[i][j] = newColor
            visited.append((i, j))
            legalNeigh = neighbor(i, j, worklist, visited)
```

```python
            worklist.extend(legalNeigh)
            worklist = worklist[1:]

    ## validity check
    if not (0 <= i < len(paint) and 0 <= j < len(paint[0])) or paint[i][j] != oldColor:
        print('cannot proceed')
        return
    wL = [(i, j)]
    vL = []
    touch(wL, vL)
```

– recursively

```python
def smudgeR(paint, i, j, oldColor, newColor):
    ## local painting function
    def touch(x, y, visited):
        for i in [x - 1, x, x + 1]:
            if i < 0 or i >= len(paint):
                continue
            for j in [y - 1, y, y + 1]:
                if j < 0 or j >= len(paint[0]):
                    continue
                if paint[i][j] != oldColor:
                    continue
                if (i, j) in visited:
                    continue
                paint[i][j] = newColor
                visited.append((i, j))
                touch(i, j, visited)

    ## validity check
    if not (0 <= i < len(paint) and 0 <= j < len(paint[0])) or paint[i][j] != oldColor:
        print('cannot proceed')
        return
    touch(i, j, [])
```

8.2 Tutorial: Multi-dimensional Arrays

Matrix – can represented by a list of lists

e.g.

A 4×10 matrix

i.e.

```python
>>> pprint(m)
[[1, 1, 1, 0, 1, 0, 0, 1, 0, 1],
 [1, 0, 1, 0, 0, 0, 1, 0, 0, 0],
 [0, 0, 1, 1, 0, 0, 0, 1, 1, 0],
 [0, 1, 1, 1, 1, 0, 0, 0, 1, 1]]
```

9 Lecture 09 (Search and Sort)

9.1 Searching & Sorting

Searching – finding something in the list

Linear Search

↪ go through the list from start to end

e.g.

```python
## equivalent code
for i in [5, 2, 3, 4]:
    if i == 3:
        return True
```

```python
def linear_search(value, lst):
    for i in lst:
        if i == value:
            return True
    return False
```

– order of growth; number of comparisons[14] made during search[15] –

Time Complexity

$$O(n) \text{ where } n \text{ is } \texttt{len(list)} \tag{28}$$

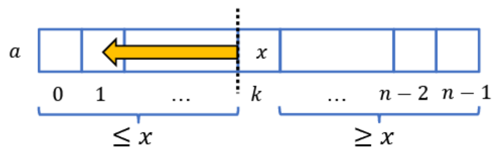
Binary Search

↪ "divide-and-conquer" sorted lists

e.g.

Assuming a list is sorted in ascending order

- if the k^{th} element is larger than what we are looking for, then we only need to search in the indices $x < k$;
- if the k^{th} element is *smaller*, then we only need to search in the indices $x > k$



- performs much better than Linear Search on average in large sorted lists
- looking for a key in a sorted list
  1. Find the middle element
  2. If it is what we are looking for, return `True`
  3. (a) If our key is smaller than the middle element, repeat search on the left of the element
     (b) Else, repeat search on the right of the element

[14] checks if the iterated element is what we want

[15] "Big-O" notation – order of n

e.g.

```python
def binary_search(key, seq):
    if seq == []:
        return False
    mid = len(seq) // 2
    if key == seq[mid]:
        return True
    elif key < seq[mid]:
        return binary_search(key, seq[:mid])
    else:
        return binary_search(key, seq[mid + 1:])
```

* *creates intermediary list through slicing – slower*

```python
def binary_search(key, seq):    ## seq is sorted
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2 ## get middle
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid - 1)
        else:
            return helper(mid + 1, high)

    return helper(0, len(seq) - 1)
```

* *searches through list through indexing – faster*

- every call to **helper** eliminates the problem size by half
  ↪ problem size gets reduced to 1 very quickly
- order of growth; number of comparisons made during searching –

Time Complexity

$$O(\log n) \tag{29}$$

Sorting – ordering all the objects in a sequence

Selection Sort

- iterate through the sequence to find the smallest element in the sequence and put it at the end of the sequence

  e.g.

```python
a = [4, 12, 3, 11]
sort = []
while a:    ## a is not []
    smallest = a[0]
    for element in a:
        if element < smallest:
            smallest = element
    a.remove(smallest)
    sort.append(smallest)
```

```
>>> print(a)
[4, 12, 3, 11]
[4, 12, 11]
[12, 11]
[12]
[]
>>> print(sort)
[3, 4, 11, 12]
```

– order of growth; number of comparisons made during sorting –

Time Complexity

$$\begin{aligned}\sum_{i=0}^{n} i &= n + (n-1) + (n-2) + \cdots + 1 + 0 \\ &= \frac{n(n+1)}{2} \\ &= \frac{n^2+n}{2} \\ &= \frac{1}{2}n^2 + \frac{n}{2} \\ &\approx n^2\end{aligned} \tag{30}$$

$$\implies \begin{cases} \textsf{Worst} & O\left(n^2\right) \\ \textsf{Average} & O\left(n^2\right) \\ \textsf{Best} & O\left(n^2\right) \end{cases} \tag{31}$$

Space Complexity

$$O(1) \tag{32}$$

* *if in-place update is ensured*
** *where n is the length of the sequence*

Merge Sort

– split sequence into half recursively until each nested "half" only has 1 element then combine each "half" back into a sorted sequence
  – Base case: `n < 2`, `return lst`
  – Otherwise:
    1. Divide list into two
    2. Sort each of them
    3. Merge!
       (a) Compare first element
       (b) Take the smaller of the two
       (c) Repeat until no more elements

```
def merge_sort(lst):
    if len(lst) < 2:     ## Base case!
        return lst
    mid = len(lst) // 2
    left = merge_sort(lst[:mid])    ## sort left
    right = merge_sort(lst[mid:])   ## sort right
    return merge(left, right)

def merge(left, right):
    results = []
    while left and right:
        if left[0] < right[0]:
            results.append(left.pop(0))
```

```python
        else:
            results.append(right.pop(0))
    results.extend(left)
    results.extend(right)
    return results
```

– order of growth; number of comparisons made during sorting –

Time Complexity

$$\left.\begin{array}{ll} \text{Worst} & O(n \log n) \\ \text{Average} & O(n \log n) \\ \text{Best} & O(n \log n) \end{array}\right\} \tag{33}$$

Space Complexity

$$O(n) \tag{34}$$

Sort Properties

– **In-place**: uses a small, constant amount of extra storage space

Space Complexity

$$O(1) \tag{35}$$

– **Selection Sort**: No (Possible)
 – **In-place Selection Sort**

```python
def selection_sort(seq):
    for i in range(len(seq)):
        ## find the location of the minimum element in the
        ↪ unsorted sequence
        min_idx = i
        for j in range(i + 1, len(seq)):
            if seq[j] < seq[min_idx]:
                min_idx = j
        ## place the minimum element from the unsorted sequence
        ↪ on the position indexed by i
        seq[i], seq[min_idx] = seq[min_idx], seq[i]
```

– **Merge Sort**: No (Possible)

– **Stability** – maintains the relative order of items with equal keys
i.e.
 values

– **Selection Sort**: Yes
 – **In-place Selection Sort**

```python
def selection_sort_stable(seq):
    for i in range(len(seq)):
```

```python
        ## find the location of the minimum element in the
        ↪  unsorted sequence
        min_idx = i
        for j in range(i + 1, len(seq)):
            if seq[j][0] < seq[min_idx][0]:
                min_idx = j
        ## place the minimum element from the unsorted sequence
        ↪  on the position indexed by i
        seq[i], seq[min_idx] = seq[min_idx], seq[i]
```

```python
>>> lps
[(19, 61), (3, 78), (20, 51), (12, 52), (12, 54), (3, 75)]
>>> selection_sort_stable(lps)
[(3, 78), (3, 75), (12, 52), (12, 54), (19, 61), (20, 51)]
```

- **Merge Sort**: No (can be)

10 Lecture 10 (Object-Oriented Programming)

Major Programming Paradigms

- **Imperative Programming**
  - *instructs computer how to function*

  e.g.

  C, Pascal, Algol, Basic, Fortran
- **Functional Programming**
  - *focus on using functions to solve problems*
  - primarily written using functions
  - every data structure is immutable

  e.g.

  Scheme, ML, Haskell
- **Logic Programming**
  - *defining & manipulating constraints to find solutions*

  e.g.

  Prolog, CLP
- **Object-oriented Programming**
  - *describing properties of objects and their interactions to solve problems*
  - very verbose
  - modularizes a problem into smaller components

  e.g.

  Java, C++, Smalltalk

10.1 Object-Oriented Programming

Conventional Way of Code & Data Organization

- using functions to organize code

  i.e.

```python
def foo(x, y):
    ...
```

- using built-in types to organize data

  i.e.

```python
[
    [ j for j in range(n) ]
for i in range(m) ]
```

e.g.

Representing a point in two-dimensional space and finding the distance from the point to another point

- `tuple` as its type

  i.e.

  `(x, y)` where x, y $\in \mathbb{R}$

  e.g.

```python
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return sqrt((x2 - x1) ** 2 +
    ↪  (y2 - y1) ** 2)
```

- `dict` as its type

  i.e.

```python
{'x': x, 'y': y}
```

  e.g.

```python
def distance(p1, p2):
    x_dist = p2['x'] - p1['x']
    y_dist = p2['y'] - p1['y']
    return sqrt(x_dist ** 2 +
    ↪  y_dist ** 2)
```

Object-oriented Programming – using *programmer-defined types* to organize both code and data

- Pros
  - simplification of complex, possibly hierarchical structures
  - easy reuse of code
  - easy code modifiability
  - intuitive methods
  - hiding of details through message passing and polymorphism
- Cons
  - overhead associated with the creation of classes, methods and instances

Class Definition

- blueprint of objects (data) of a class (type)
- factory for creating objects/instances of a class (type)

e.g.

Introducing a new type `Point` to capture both data & code

i.e.

```
class Point():
    ''' Represents a point in 2-D space.
    Attributes: x, y '''
    def __init__(self, x, y):
        ''' construct a point '''
        self.x = a
        self.y = b

    def __str__(self):
        return 'Point at: x: {}, y: {}'.format(self.x, self.y)

    def distance(self, pt):
        ''' compute distance between this point and point pt '''
        x_dist = self.x - pt.x
        y_dist = self.y - pt.y
        return sqrt(x_dist ** 2 + y_dist ** 2)
```

- `Point` is a newly-defined type, not a piece of data; formally called a `class`
- `class Point()`[16] – defines a class `Point`[17]
- `def __init__(self, ...):` – construction function
  - specifies how an object/instance of a class is to be created and initialized
  - special method[18]
    - built-in in Python; specifies to Python that it is to be handled differently
  - not to be called directly; called by calling the class name
  - directly assigning values to parameters inside the parentheses after `__init__` sets the default values of those parameters

    e.g.

    ```
    def __init__(self, hr = 0, mn = 0, sec = 0):
    ```
- `self` – reference to the object itself
  - allows references to its attributes and calling of methods in the class
  - attributes assigned to self

    i.e.

    `self.x` or `self.y`
  - first parameter in method definitions in a class
  - not passed to method during method calls in normal way

[16] `class` is a Python keyword

[17] By convention first character of class name is capitalized

[18] Any method having '`__`' before and after a name is considered special methods of their respective class

- `def __str__(self):` – returns a string representation of an object
  - special method

  e.g.
  ```
  >>> p1 = Point(2, 5)
  >>> str(p1)
  'Point at: x: 2, y: 5'
  >>> print(p1)
  Point at: x: 2, y: 5
  ```
- subsequent method definitions – defines actions associated with class objects

Instantiation – creating an instance (data object) of the class

- instances of classes can be created and assigned to variables

  e.g.
  ```
  >>> p1 = Point(3, 4)
  >>> p2 = Point(4, 3)
  >>> type(p1)
  <class '__main__.Point'>
  >>> type(p2)
  <class '__main__.Point'>
  ```
- dot notation is used to access attributes and methods of the object

  e.g.
  ```
  >>> p1.x
  3
  >>> p1.y
  4
  >>> p1.distance(p2)
  1.4142135623730951
  ```
- instances are first-class citizens in Python
  - can pass instances as arguments to a function
  - can return instances as a function result

    e.g.
    ```
    def mid_point(p1, p2):
        x_coord = (p1.x + p2.x) / 2
        y_coord = (p1.y + p2.y) / 2
        mpt = Point(x_coord, y_coord)
        return mpt
    ```
    ```
    >>> p1 = Point(3, 4)
    >>> p2 = Point(4, 3)
    >>> mp = mid_point(p1, p2)
    >>> mp
    <__main__.Point object at 0x0486DFD0>
    >>> mp.x
    4.5
    >>> mp.y
    6.0
    ```
  - can store instances in a sequence
  - can assign instances as an attribute of another class instance
    - can be accessed via nested references of attributes

      e.g.
      ```
      self.corner.x
      ```

Methods[19] – a function associated with a particular class

- defined inside a `class` definition
- first parameter of a method is `self`
- syntax for invoking a method is different from normal function call

  e.g.

  ```
  >>> p1 = Point(2, 4)
  >>> p2 = Point(12, -4)
  ```

  - Function call syntax
    - *treats* `class` *like a library and uses method as a function*

    i.e.

    ```
    >>> Point.distance(p1, p2)
    >>> 12.806248474865697
    ```

    * *uses* `class` *name*
  - Method call syntax

    i.e.

    ```
    >>> p1.distance(p2)
    >>> 12.806248474865697
    ```

    * *uses instance name*
- `__add__(self, ...)` method – operator overloading
  - operator overloading – changing the behavior of an operator (e.g. "+") so that it works with programmer-defined types

  e.g.

```
class Time():
    ''' Represent the time of day '''
    def __init__(self, hr = 0, mn = 0, sec = 0):
        self.hour = hr
        self.minute = mn
        self.second = sec

    def __str__(self):
        return '{:2d}:{:2d}:{:2d}'.format(self.hour, self.minute, self.second)

    def increment(self, seconds):
        seconds += self.to_int()
        return int_to_time(seconds)

    def add_time(self, other):
        seconds = self.to_int() + other.to_int()
        return int_to_time(seconds)

    def to_int(self):
        mns = self.hour * 60 + self.minute
        secs = mns * 60 + self.second
        return secs

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)
```

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20: 0
```

```
>>> elapsed = duration.to_int()
>>> elapsed
5700
>>> print(start + elapsed)
11:20: 0
```

↪ `__add__(self, other)` – checks the `type` or `other` and invokes either `add_time` or `increment` method

[19] semantically the same as functions, but with syntactic differences

- **Polymorphism** – function that work with several types
    - facilitates code reuse

  e.g.

```
>>> str_lst = ['adam', 'loves', 'eve']
>>> int_lst = [10, 20, 30]
>>> print(reduce(lambda x, y : x + y, int_lst))
60
>>> print(reduce(lambda x, y : x + y, str_lst))
adamloveseve
```

```
>>> t1 = Time(1, 2, 3)
>>> duration = 214
>>> t2 = t1 + duration
>>> t3 = t2 + duration
>>> time_list = [t1, t2, t3]
>>> print(reduce(lambda x, y : x + y, time_lst))
 3:16:51
```

  * *type-based dispatch is useful when necessary*
  ** *functions should work correctly for arguments with different types without explicitly checking for the* `type`

Class Inheritance

- inheriting information from another class
- common attributes/methods between classes can be extracted into and inherited from the same base class[20]

  e.g.

```
class Vehicle:
    def __init__(self, pos):
        self.pos = pos
        self.velocity = (0, 0)

    def setVelocity(self, vx, vx):
        self.velocity = (vx, vy)

    def move(self):
        self.pos = (self.pos[0] + self.velocity[0], self.pos[1] + self.velocity[1])
        print(f'Move to {self.pos}')

class Sportscar(Vehicle):
    def turnOnTurbo(self):
        print('VROOOOOOOM......')
        self.velocity = (self.velocity[0] * 2, self.velocity[1] * 2)
        print(f'Velocity increased to {self.velocity}')

class Lorry(Vehicle):
    def __init__(self, pos):
        super().__init__(pos)
        self.cargo = []

    def load(self, cargo):
        self.cargo.append(cargo)

    def unload(self, cargo):
        if cargo in self.cargo:
            self.cargo.remove(cargo)
            print(f'Cargo {cargo} unloaded.')
        else:
```

[20] called the superclass

```python
            print(f'Cargo {cargo} not found.')

    def inventory(self):
        print('Inventory: ' + str(self.cargo))

class Bisarca(Lorry):
    def load(self, cargo):
        if isinstance(cargo, Vehicle):
            super().load(cargo)
        else:
            print(f'Your cargo ({cargo}) is not a vehicle!')
```

```
>>> myCar = Sportscar((0, 0))
>>> myCar.setVelocity(0, 40)
>>> myCar.move()
Move to (0, 40)
>>> myCar.turnOnTurbo()
VROOOOOOOM......
Velocity increased to (0, 80)
>>> myCar.move()
Move to (0, 120)
```

```
>>> myTruck = Lorry((10, 10))
>>> myTruck.setVelocity(10, 0)
>>> myTruck.move()
Move to (20, 10)
>>> myTruck.load('Food')
>>> myTruck.load('Supplies')
>>> myTruck.inventory()
Inventory: ['Food', 'Supplies']
>>> myTruck.unload('Food')
Cargo Food unloaded.
>>> myTruck.inventory()
Inventory: ['Supplies']
>>> myTruck.unload('Gold')
Cargo Gold not found.
>>> myDadTruck = Bisarca((0, 0))
>>> myDadTruck.load('Food')
Your cargo (Food) is not a vehicle!
>>> myDadTruck.load(myCar)
>>> myDadTruck.load(myTruck)
>>> myDadTruck.inventory()
Inventory: [<__main__.Sportscar object at 0x10d3ecd50>, <__main__.Lorry object at 0x10d39dc10>]
```

↪ `Sportscar` inherits EVERYTHING from `Vehicle`

↪ `__init__(self, ...)` method is reused from its parent

- subclass – the class that inherits from another class
  e.g.

  ```
  >>> issubclass(Lorry, Vehicle)
  True
  ```

  - `Vehicle` – super/parent class
  - `Sportscar`/`Lorry` – subclass/child class

  ↪ usually a more specific type of its parent class
- `isinstance(<obj>, <class>)` – checks if an instance `<obj>` belongs to a class or is a subclass of a certain class `<class>`

Adding Attributes to `__init__(self, ...)`

1. **Overriding**
   - completely redefines the method with the same name that was in the parent class
   - overrides the method inherited from its parent class; subclass will call the new redefined method instead of the parent's method

   e.g.

```
class Lorry(Vehicle):
    def __init__(self, pos):
        self.pos = pos
        self.velocity = (0, 0)
        self.cargo = []
```

2. **Calling `super()` class**
   - redefines the method while calling the method in parent class in the method definition

   e.g.

```
class Lorry(Vehicle):
    def __init__(self, pos):
        super().__init__(pos)
        self.cargo = []
```

   ↪ way to access method in parent/higher classes
   ↪ "functionally" creates a temporary object of the parent class

   * *can treat a call to a parent method through '`super()`' as directly "copying and pasting" the method definition being called into the new method definition*

- calling the `super()` class is preferred
  ↪ no duplication of code

Multiple Inheritance

- call constructors from both parent classes

  e.g.

```
class Cannon:
    def __init__(self):
        self.numAmmo = 0

    def fire(self):
        if self.numAmmo:
            print('Fire!!!!!!!')
            self.numAmmo -= 1
        else:
            print('No more ammo')

    def reload(self):
        if self.numAmmo:
            print('Unable to reload')
        else:
            print('Cannon related')
            self.numAmmo += 1

class Tank(Vehicle, Cannon):
    def __init__(self, pos):
        Vehicle.__init__(self, pos)
        Cannon.__init__(self)
```

```
>>> myCannon = Cannon()
>>> myCannon.fire()
No more ammo
>>> myCannon.reload()
Cannon reloaded
>>> myCannon.reload()
Unable to reload
>>> myCannon.fire()
Fire!!!!!!!
>>> myCannon.fire()
No more ammo
>>> myTank = Tank((0, 0))
>>> myTank.setVelocity(40, 10)
>>> myTank.move()
Move to (40, 10)
>>> myTank.move()
Move to (80, 20)
>>> myTank.reload()
Cannon reloaded
>>> myTank.fire()
Fire!!!!!!!
```

  ↪ calls `__init__(self, ...)` in each parent class as functions

Resolving Methods

- when calling a method that exists in the parent class of an instance and its parent class, the nearest method will be called
  e.g.

```
class BattleBisarca(Bisarca, Cannon):
    def __init__(self, pos):
        Bisarca.__init__(self, pos)
        Cannon.__init__(self)
```

```
>>> OptimasPrime = BattleBisarca((0, 0))
>>> OptimasPrime.load('Food')
Your cargo (Food) is not a vehicle!
```

- complication arises when the same method is available in two distinct superclasses
  - *Diamond Inheritance*
- not supported in many OOP languages
  i.e.
  C++, Java
- causes more trouble sometimes due to unexpected method calls in complicated inheritance structures
- only use MI if the parent classes are very clearly different!

Private v.s. Public

- **Private Methods**
  - function that can only be used inside your class but cannot be called outside
  - prevents any programmers from accessing the method/variable in a wrong way
    e.g.
    Directly changing the balance of a bank account
    e.g.

```
myAcc.balance = 100000000
```

  - a class method can be made private by adding two underscore '__' before the method name
    e.g.[21]

```
class Bisarca(Lorry):
    def __convertCargo(self):
        output = []
        for c in self.cargo:
            output.append(str(type(c)).split('.')[1].split('\'')[0])
        return output

    def inventory(self):
        print('Inventory: ' + str(self.__convertCargo()))
```

```
>>> myDadTruck.__convertCargo()
AttributeError: 'Bisarca' object has no attribute '__convertCargo'
```

  - in other OOP languages (e.g. C++, Java), a private method/variable will NOT be accessible by anyone other than the class itself; Python does not have that "full protection"
  - in Python, private methods can still be accessed by adding '_' and the class name in front of the method name
    e.g.

```
>>> myDadTruck._Bisarca__convertCargo()
['Sportscar', 'Lorry']
```

[21] \– denotes an escape character

11 Notes

Floating-point Error

e.g.

```
>>> distance(p1, p2)    ## = 49.9999
>>> distance(p1, p2) == 50
False
```

– avoid floating-point numbers in integer comparisons!
  ↪ convert floating point value comparisons into an integer value comparison

"Paired" Comparisons

– when multiple comparison operations are chained, interpret it as paired `and` comparisons
  e.g.
  `5 > 3 > 4 == False` as `(5 > 3) and (3 > 4) and (4 == False)`

How fast is my desktop?

– creates 100M of numbers, estimated to be 400MB of data

```
import time
def create_list(n):
    start_time = time.time()
    l = [i for i in range(n)]
    end_time = time.time()
    print('Duration = ' + str(end_time - start_time) + ' s')
```

```
>>> create_list(1000)
Duration = 0.0 s
>>> create_list(1000 * 1000)
Duration = 0.04769182205200195 s
>>> create_list(1000 * 1000 * 100)
Duration = 6.026865720748901 s
```

* *`time.time()` – returns the time in seconds since the epoch (1970/1/1 00:00) as a floating point number*

Understanding Python's Slice Assignment[22]

For all examples in this section, `nums = [1, 2, 3, 4, 5]`

– **Basic Syntax**

In order to understand Python's slice assignment, you should at least have a decent grasp of how slicing works. Here's a quick recap:

```
[start_at:stop_before:step]
```

Where `start_at` is the index of the first item to be returned (included), `stop_before` is the index of the element before which to stop (not included) and `step` is the stride between any two items.

[22] https://www.30secondsofcode.org/python/s/slice-assignment/

Slice assignment has the same syntax as slicing a list with the only exception that it's used on the left-hand side of an expression instead of the right-hand side. Since slicing returns a list, slice assignment requires a list (or other iterable). And, as the name implies, the right-hand side should be the value to assign to the slice on the left-hand side of the expression. For example:

```
>>> nums[:1] = [6]       ## (replace elements 0 through 1)
>>> nums
[6, 2, 3, 4, 5]
>>> nums[1:3] = [7, 8]   ## (replace elements 1 through 3)
>>> nums
[6, 7, 8, 4, 5]
>>> nums[-2:] = [9, 0]   ## (replace the last 2 elements)
>>> nums
[6, 7, 8, 9, 0]
```

– **Changing Length**
The part of the list returned by the slice on the left-hand side of the expression is the part of the list that's going to be changed by slice assignment. This means that you can use slice assignment to replace part of the list with a different list whose length is also different from the returned slice. For example:

```
>>> nums[1:4] = [6, 7]       ## (replace 3 elements with 2)
>>> nums
[1, 6, 7, 5]
>>> nums[-1:] = [8, 9, 0]    ## (replace 1 element with 3)
>>> nums
[1, 6, 7, 8, 9, 0]
>>> nums[:1] = []            ## (replace 1 element with 0)
>>> nums
[6, 7, 8, 9, 0]
```

If you take empty slices into account, you can also insert elements into a list without replacing anything in it. For example:

```
>>> nums[2:2] = [6, 7]   ## (insert 2 elements)
>>> nums
[1, 2, 6, 7, 3, 4, 5]
>>> nums[7:] = [8, 9]    ## (append 2 elements)
>>> nums
[1, 2, 6, 7, 3, 4, 5, 8, 9]
>>> nums[:0] = [0]       ## (prepend 1 element)
>>> nums
[0, 1, 2, 6, 7, 3, 4, 5, 8, 9]
>>> nums[:] = [ 4, 2]    ## (replace whole list with a new one)
>>> nums
[4, 2]
```

– **Using Steps**
Last but not least, `step` is also applicable in slice assignment and you can use it to replace elements that match the iteration after each stride. The only difference is that if `step` is not `1`, the inserted list must have the exact same length as that of the returned list slice. For example:

```
>>> nums[2:5:2] = [6, 7]     ## (replace every 2nd element, 2 through 5)
>>> nums
[1, 2, 6, 4, 7]
>>> nums[2:5:2] = [6, 7, 8] ## (can't replace 2 elements with 3)
ValueError
>>> nums[1::-1] = [9, 0]     ## (reverse replace, 1 through start)
>>> nums
[0, 9, 6, 4, 7]
```