CMPUT 175: introduction to the foundations of computation II

algorithm: ordered list of instructions

- without an algorithm, there can be no program

programming = data + algorithms

programming = data structures + pseudo-code algorithms

- translated into a programming language which is compiled or interpreted into machine code

PYTHON REVIEW

- values and types:
  - int — integer
    - positive or negative number without decimals
    - can be as long as how much memory you have on your computer
    - can do all normal mathematical operations and some extras
    - +, -, **, /, *, ***//*, **%**
  - float
    - finite approximations of real numbers (numbers with a decimal point)
    - can do all normal mathematical operations
    - +, -, **, /, *

type() — check type of object

str — string

- quotes on either side of string (’’ or “”)
- can do iterable indexing
- operations on strings are case sensitive
- can do concatenation using +
- immutable

len() — check length of iterable object

<str>.lower() — convert string to lowercase

<str>.upper() — convert string to uppercase

None — value expressing absence of a value

▼ variables:

```
# grocery shopping example

# sub total
1 * 1.78 + 0.77 + 1.29 + 5.99
# calculate taxes
0.05(1 * 1.78 + 0.77 + 1.29 + 5.99)
# add to total
0.05(1 * 1.78 + 0.77 + 1.29 + 5.99) + 1 * 1.78 + 0.77 + 1.29 + 5.99

# there must be a better way... VARIABLES!

avocado = 1.78 # assignment statement
lemon = 0.77
onion = 1.29
oregano = 5.99

subtotal = 0
subtotal = 2 * avocado # reassignment
subtotal += lemon
subtotal += onion
```

```
subtotal += oregano

tax = 0.05 * subtotal

total = subtotal + tax

# get to right decimal points using string formatting
print("%0.2f" % total)
```

variables — allow us to assign a value to a name (identifier)

- case sensitive
- cannot start with a digit
- cannot contain spaces
- cannot have hyphens
- cannot be an existing keyword

camelcase — ex: HelloMyNameIsMegan

help('keywords') — gives you list of keywords

▼ if statements:

```
if <condition>:
  statement(s)
elif <condition>: # not always necessary
  statements
else: # not always necessary
  statement(s)
```

```
# receipt example

subtotal = 80

if subtotal >= 300:
  discount = subtotal * 0.05
```

```
elif subtotal >= 100:
  discount = suntotal * 0.02
else:
  discount = 0
```

condition — anything that returns a bool

not for negation

and for conjunction

or for disjunction

if statements create control flow

▼ for loops:

for loops — iterating over something a fixed number of times

```
# dog example
for num in range(5): # num is a variable
  print("BORK!") # body

# what is num assigned to?
for num in range(3):
  print("num is ", num)
  print()
# num is reassigned for every iteration of the loop

#another example
string = "CMPUT175"
for num in range(len(string)):
  print(string[nun])
```

range() — returns an iterable

```
# for loop iterating without range()
string = "CMPUT175"
for character in string:
  print(character)
```

```
# another example

word = input("enter a word: ")
vowels = 0
consonants = 0

for letter in word:
  if letter in "aeiou":
    vowels += 1
  else:
    consonants += 1

ratio = consonants / vowels

print(word, "has", ratio, "consonants to vowels")
```

▼ while loops:

while loops — useful when not sure how many iterations until loop stops

```
# example
number = 0

while number >= 0: # condition
  number = int(input("Please enter a negative number: ")) # body

print("Thank you,", number, "sure is a negative number.")
```

```
# infinite loop example
# condition will never be False
shouldAskQuestion = True

while shouldAskQuestion:
  number = int(input("Please enter a negative number: "))

print("Thank you,", number, "sure is a negative number.")
```

▼ tuples and lists:

tuples — allow you to group values together in order

surrounded by ()

immutable

allow us to quickly group items together

lists — store items in order

surrounded by []

mutable

.sort() — sort list items in alphabetical/ ascending order

.append(<item>) — add item to end of list

.pop() — returns last item and removes item from list

```
# game librayr example

library = [
  ('Super Metroid', 'Intelligent Systems', 1994, 1994),
  ('ActRaiser', 'Quintet', 1990, 1991),
  ('EarthBound', 'APE', 1994, 1995),
  ('ChronoTrigger', 'Square', 1995, 1995)
]

library.sort() # only rearranges list items, not what the list items contain

library.append(('Plok!', 'Software Creations', 1993, 1993))
library.append(library[1]) # can double add items
```

▼ sets:

set — unordered collection of unique values

surrounded by {}

can use operators

mutable

never contains duplicate values

```
# pet example

cute = {'cat', 'dog', 'iguana', 'ferret'}
cuddly = {'cat', 'dog', 'ferret'}
eat_veg = {'dog', 'iguana'}
eat_meat = {'cat', 'dog', 'snake', 'ferret'}
laid_back = {'cat', 'snake', 'iguana'}

# set intersection
# returns items that are in both sets
cute & eat_meat

# set union
# returns items that in the first or second set
cuddly | laid_back

# set difference
# returns items of the first set that are not in the second set
eat_meat - eat_veg

# subset relation
# returns a bool
# whether the first set is completely within the second set
cuddly.issubset(cute)

# mutability
cuddly.add('snake')
cute.remove('cat')

# even though you add multiple times,
#'snake' will only appear opnce in the cuddly set
cuddly.add('snake')
cuddly.add('snake')
cuddly.add('snake')
```

len() — get length of set

▼ files:

```
# file reading example
file = open("excerpt.tect", "r") # r is the mode
```

```
file.read()

file.close() # remember to do this after you fulyly read the fil

# another reading example
file = open("filename.txt", "r")
original = file.readline()

# get rid of \n
originial.strip()

file.close()
```

```
# file writing example
file = open("hello.txt", "w") # w truncates in the file already exists (overwrites ol
d file)

file.write("hello\n")
file.write("world\n")

# must do this for file to write on computer
file.close()
```

```
# file append example
file = open("hello.txt", "a") # a adds text to the end of the file that already exist
s

file.write("goodbye\n")
file.write("world\n")

# must do this for file to write on computer
file.close()
```

\n — newline character

▼ dictionaries:

dictionary — container where elements are indexed by keys

set of key value pairs

surrounded by {}

key value pairs separated by :

unordered

can access value using corresponding key

```
# student example

students = {
  '1234':'Paul', '5678':'Peng',
  '9876':'Yasmin', '5432':'Maria'
}

students['5678'] # 'Peng'
students['6543'] # 'Maria'

students.keys() # get all the keys in the dictionary
students.values() # get all the values in the dictionary
```

values — can be any type of object, can be duplicates of other keys' values

keys — have to be unique, value can be anything

▼ string formatting:

```
# example

field = "CMPUT"
num = "175"

# print them together on one line
print(field + num) # str concatenation

# print with space in between on one line
print(field, num) # , adds default separator of a space

# print with separator other than space
print(field, num, sep = "-")
```

```
# print on spearate lines in commands one after the other
print(field); print(num)

# redefine the ending of the print function
print(field, end = " #"); print(num) # prints on same line with " #" at the end of th
e first function
```

placeholders — where to place multiple object types in a string

placeholders prefixed by %

“<string with placeholders>” % (<variables or values>)

```
# another example

field = "CMPUT"
num = "175"
stuName = "Eddie"
mark = 91.66666

# placeholder use
# can also not use variables and just print the right hand side of the equation
s = "%s receieved %5.2f in %s-%d" % (stuName, mark, field, num)
s # Eddie received 91.67 in CMPUT-175
```

modifiers — defined extra conditions of a placeholder

```
# modifier examples

# number modifier
%10d # field width

# - modifier
%-10d # left justified in field width

# 0 modifier
%010d # pre-fill with leading zeros
```

```
# . modifier
%10.2f # decimal points
```

```
# modifier examples cont

field = "CMPUT"
num = "175"
stuName = "Eddie"
mark = 91.66666

print("[%10s]" % (field)) # [     CMPUT]
print("[%-10s]" % (field)) # [CMPUT    ]

print("[%010.2f]" % (mark)) # [0000091.67]
```

▼ jan7 lecture

program — sequence of instructions that specified how to perform a computation

access data (input), process data (basic operations, conditional operations, repeated operations), provide results (output

object orientation — how the python programs works

python is also dynamic; variables exist automatically int he first scope where they are assigned

class — template that describes the properties and behaviour of a group of objects

behaviour implemented using methods

object — instance of a class

stored in main memory

has an id, value, type (shape and format... type determines kind of operations you can perform)

sometimes object are referred to as vaues

can be stored, accessed and manipulated via identifiers (variables)

variables — names referring to objects/values (aka identifiers)

a value us assigned to a variable

should start with a letter

can contain _

case sensitive

cannot be keywords

descriptive identifiers are preferred and are more informative

naming conventions;

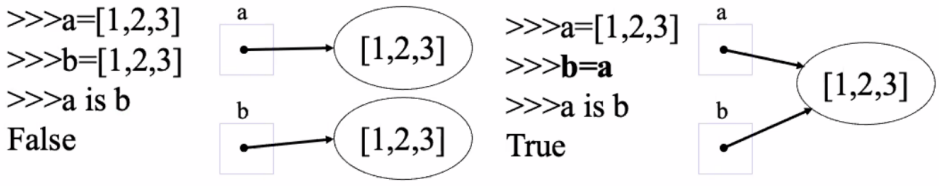
snake_case

lowerCamel

UpperCamel

aliasing — the same object is being referred with more than one variable

the two variables have the same memory address



careful with mutable objects, since they can be changed without reassignment, so both variables of the aliased object would refer to the same changed object

this is not the case with immutable objects, instead a whole new object would come into existence and the two variable would not refer to the same object

▼ jan10 lecture

aliasing can cause problems

types and operators — useable operators depend on object type

integer and float — int, float, long, complex

+, -, , /, //, %, Z*

boolean;

and, or, not for conjunction, disjunction, and negation

comparison operators; ==, !=, <, >, <=, >=

str — immutable sequence of characters

+ for concatenation * for repetition

sequence operations; methods in the string class

count(item), find(item), split(char)

lower(), upper()

center(w, char), ljust(w), rjust(w)

strip(), replace(old, new, max)

```
help(str) # gives list of all methods in str class

str.lower() # returns COPY of str in lowercase
str.upper() # returns COPY of str in uppercase

str.find(char) # finds character in str and returns int value of characte
r's index

str.strip() # removes leading and trailing whitespace, can specify charact
ers
```

```
str.ljust(width, fill_value)n # returns left-justified str of length width
with padding of the fill_value
str.r_just(width, fill_value) # same thing but right-justified
str.center(width, fill_value) # same thing but centered

str.split(char) # returns a list of items split at the specified character
(does not include split character)
```

list — mutable heterogenous sequence of values of any type

+ and * for concatenation and repetition

slice using indexes (ex: alist[2:4] —> list elements at index 2 and 3)

in to check membership (ex: 3 in alist —> returns bool)

len() for length of list

also...

```
list.append(item) # sppends item at end of list

list.insert(i, item) # inserts item at ith position of list

list.pop() # removes and returns last item of list
list.pop(i) # removes and returns ith element in a list

list.del(i) # removes ith element in list

list.remove(item) # removes first occurrence of item

list.sort() # modifies list to be sorted in ascending order

list.reverse() # modifies list to be reverse order

list.count(item) # returns number of occurrences of item in list

list.index(item) # returns index of first occurrence of item
```

```
list()
# example
list("CMPUT")
# ["C", "M", "P", "U", "T"]
```

```
# methods in the string class
str.split()
#example
# creates new list object
"1,2,3,,5".split(",")
# ['1', '2', '3', '', '5']

str.join()
# example
# converts list to str
' '.join(['1', '2', '3']
# '1 2 3'
```

tuple — immutable list

mutable objects within tuples can be changed but not replaced

set — mutable unordered collection of immutable objects

objects must be unique

```
# union of sets
set1 | set2

# intersection of sets
set1 & set2

# in set1 but not set2
set1 - set2

# add to set
set1.add(immutable_object)

# return bool of subsets/supersets
set1.issubset(set2) # all set1 in set 2?
set1.issuperset(set2) # all set2 in set1?

# remove item
set1.pop() # would remove and return random item
set1.remove(item) # removes specified item from set, returns None
```

dictionary — mutable collections of unordered associated pairs of items

key:value pairs

values are accessed via keys

dict.keys() returns list of keys

dict.values() returns list of values

dict.items() returns (key, value) of dict

in checks keys

sequence — container that can hold more than one item

format method — in str class used for str formatting

THE NEED FOR DATA STRUCTURE

- data structures:

the right choice of data structure can impact code's...

- speed/efficiency
- clarity conciseness/maintainability
- use of memory

when choosing data structure, need to keep in mind;

- reusability
- ease of use and maintainability
- space required
- efficiency

data mining — field in comp sci that consists of discovering and extracting patterns from large collections of data

patterns are of interesting and actionable knowledge that decision makers use as insights to make better informed strategic decisions

▼ association rules

one pattern from data mining is called association rules

searches for relationships between items in a dataset

rule form: antecedent —> concequent (antecedent associated with concequent)

```
# example in market basket analysis

# customer wants to buy items from a store
# customer transactions are stored in a transactional database
{ID, time_stamp, bread, milk, apples, butter...} # transaction

# association rule example:
#  if someone buys bread, they will also buy milk
{'bread'} --> {'milk'} [0.6%, 65%] <-- supporting confidence
#  if someone buys bread and milk together, there is a 75% chanc that they will al
so buy butter
{'bread' and 'milk'} --> {'butter'} [0.2%, 75%]
```

only relevant associations are important

need to count the frequent ones

frequent based on some threshold

so we need to count all sets of items and check their frequencies

frequent itemset generation — given d items, there are 2^d possible candidate itemsets

▼ apriori algorithm:

apriori algorithm — ignore infrequent/rare items

scans database to look for only the unique items that are frequent enough (using threshold)

uses these to calculate candidate pairs

scan database again to find frequent pairs out of these candidates

then use frequent pairs to find candidate triplets

scan the database again to verify which triplets are frequent

and so on for larger itemsets until it cannot find larger candidates

you will need list for each itemset to access its counter (see how frequent the itemset is), which can be large and time consuming to increment through

▼ hash tables:

use of hash tables — faster and more constant accessing of the counter of an itemset

hash function gives us directly, given the itemset (x, y), where the counter for (x,y) is in a list to jump to its counter

don’t need to traverse the list over and over, instead the list is called a hash table

hash tables — called dictionaries in Python

built in data structures

▼ COFI algorithm:

COFI algorithm — find the same patterns as apriori algorithm by only scanning the database twice

first scan gives frequent one itemsets

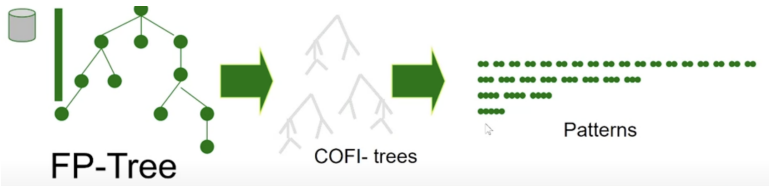
second scan we build special data structure to store relevant data

frequent parent tree (FP-Tree) — distinct structure used to compress the relevant transactions

mine this tree to consecutively create other trees

cooccurring frequent item trees (COFI-trees)

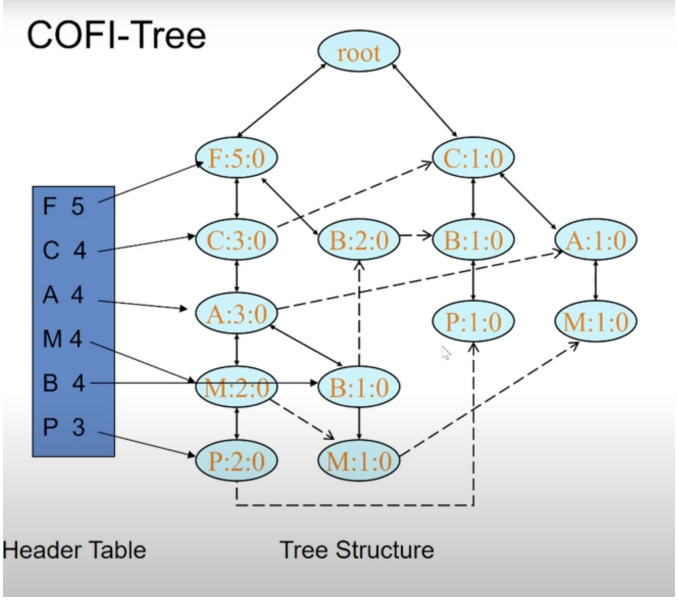
little by little discovering the patterns but in a different order



not built-in data structures

specialized data structures that are complicated but allow to do the job by scanning millions of transactions only twice (much faster)

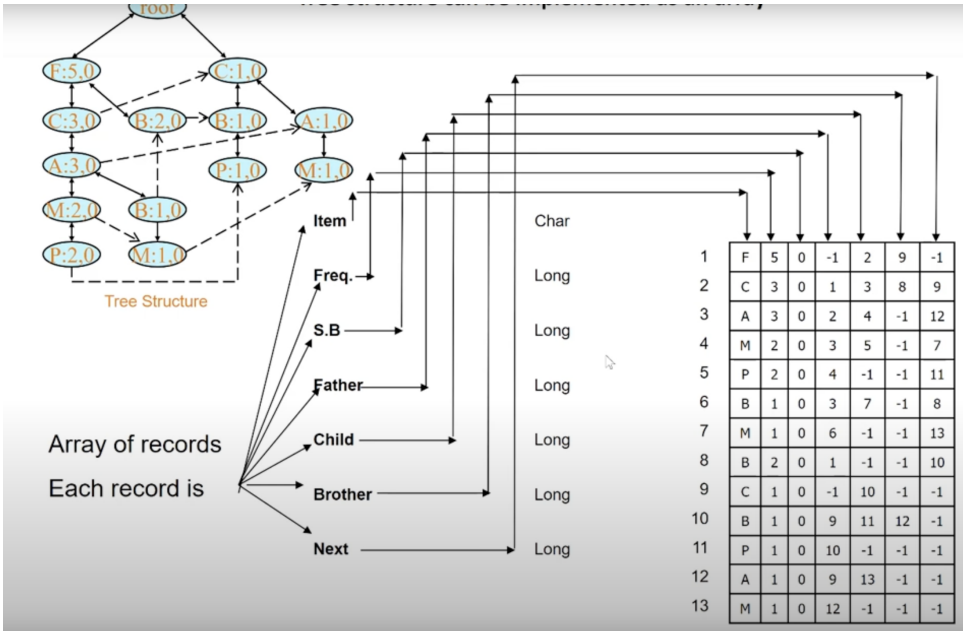
COFI-tree — combination of a table pointing to sections of a tree



not typical tree; graph with nodes of data pointing to other nodes

some are unidirectional nodes, some are bidirectional

tree structure can be implemented as an array



trees built using arrays;

pros — faster, less memory usage since there is no need to maintain any pointers

cons — size of arrays should be known in advance otherwise expensive operations could be implemented

trees built using link lists;

pros — dynamic and easily accepts the growth of the tree

cons — in genera; uses more memory and slower as it needs to allocate and maintain pointers

TRANSPOSITION CIPHER

▼ cryptography:

one approach to cryptography is the use of a transposition cipher

the order of characters is changed to obscure the message

ex: scytale, encrypted text rows from text in columns

Encrypt the message "Meet at three pm today at the usual location " using rows of 6 characters.

1	2	3	4	5	6
M	E	E	T	A	T
T	H	R	E	E	P
M	T	O	D	A	Y
A	T	T	H	E	U
S	U	A	L	L	O
C	A	T	I	O	N

The encoded message would be:

MTMASC EHTTUA EROTAT TEDHLI AEAELO TPYUON

The spaces would be removed or repositioned to hide the size of table used, since that is the encryption key in this message.

Final encryption:
MTMASCEHTTUAEROTATTEDHLIAEAELOTYPUON

Decrypt the message AES EAO OIT VUI NNN NEE RLC FTE LNV LER LS if it was encrypted using a tabular transposition cipher with rows of length 4 characters.

Since there are $10 \cdot 3 + 2 = 32$ characters and each row as 4 characters, $32/4 = 8$ means we need 8 rows.

	1	2	3	4
1	A	T	E	L
2	E	V	E	N
3	S	U	R	V
4	E	I	L	L
5	A	N	C	E
6	O	N	F	R
7	O	N	T	L
8	I	N	E	S

AT ELEVEN
SURVEILLANCE
ON FRONT LINES

since the transposition cipher does not change the frequency of individual letters...

it is still susceptible to frequency analysis

though, the transposition does eliminate information from letter pairs

▼ jan12 lecture

input() — returns value of type string

explicit type conversion required

```
var = input("Please enter string: ")
```

print() examples:

```
print('abc')
# abc
print('abc', 'def')
# abc def
print('abc' + 'def')
# abcdef
print('abc', 'def', sep = '-')
# abc-def
print('abc', 'def', end = '-')
print('ghi')
# abc def-ghi
```

str formatting old method — % as placeholder

```
print("<string expression>" % (values))

print("%s is %d years old." % (sName, age))

%d or %i --> integer
%f --> floating point
%s --> string

%15d --> field width (right justified)
%-15d --> field width (left justified)
```

```
%015d --> pre-fill wit5h leading zeros
%15.2 --> field width and decimal points
```

str formatting format method — uses .format()

```
print('hello my name is {0:^15}, I have {1:>7.2f} dollars.'.format(name, amount))
# hello my name is     megan     I have  5.3  dollars
```

don’t have to specify s and d, but must specify f

control structures:

```
# conditionals
if condition:
  statements
elif condition:
  statements
else:
  statements

# loops
while condition: # indefinite repitition
  statements

for var in sequence: # definite repitition
  statements

# continue example
a list =][
for i in alist:
  if i == 0:
    continue
  print(i)

# break example
a list =][
for i in alist:
  if i == 0:
    break
  print(i)
```

files:

```
open(filename, "") # opens file for a specific mode
f.read() # reads all contents in one
f.realines() # returns list of lines
f.write(str) # writes string in file
f.close() # closes file
f.tell() # returns the current position in file
f.seek(offset[, from]) sets the files current position at offset


# open file modes
r # read
r+ # read and write
w # write only
a # append to end of file
b # append a b for binary file
```

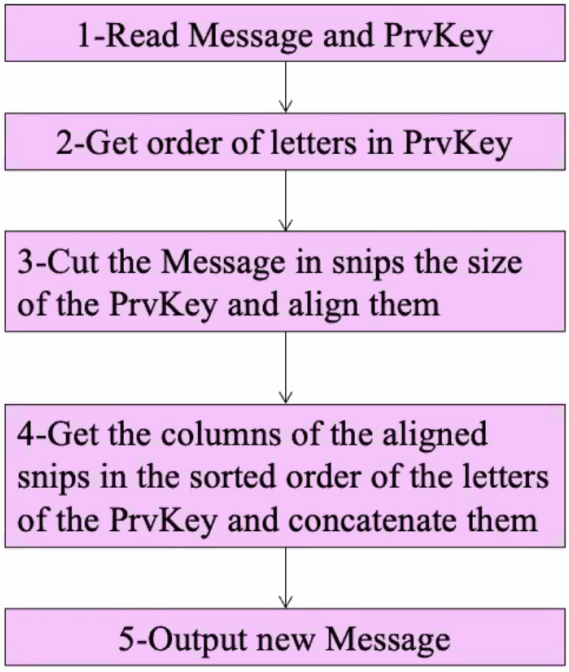
```
#reading file examples
# both produce identical outputs

# view file as list
infile = open(filename, 'r')
for line in infile:
  line = line.strip('\n')
  print(line)

# reads file into a list
infile = open(filename, 'r')
alist = infile.read().splitlines() # splits and gets rid of newline characters
for line in alist:
  print(line)
```

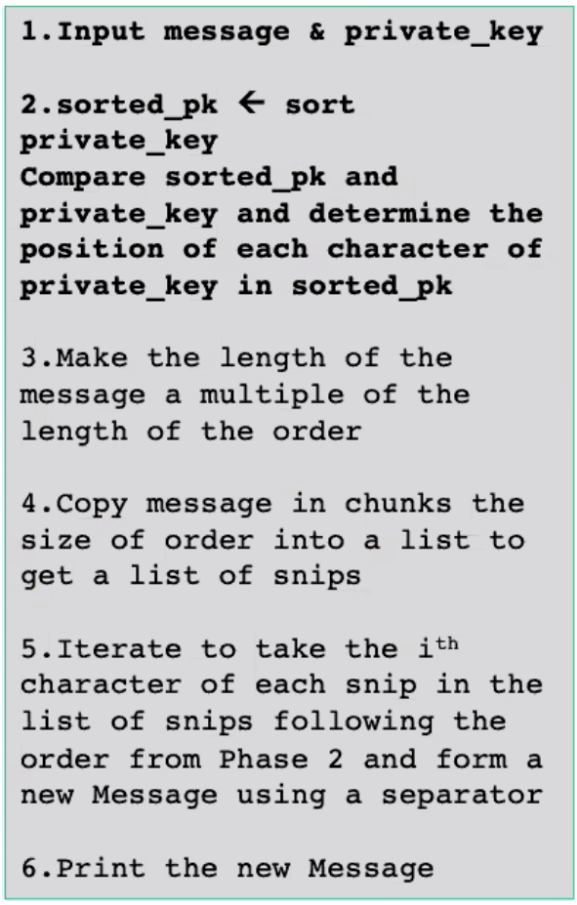
functions and procedures:

- function — returns anything but a NoneType object
- procedure — returns a NoneType object
  - just executes commands
  - also a type of function, just referred to a different name
- both can be called multiple times in a program

```
def name (arg1, arg2):
  statements
```

simple enciphering algorithm:





▼ another string formatting example

```
# String Formatting Example using the format method in the str class
name = 'Fred'
stu_id = 3456
average = 78.6666
grade = 'B'

# {<position in the tuple> : <fill_symbol><justification><field_width}
# fill symbol - can be any character to fill in the remaining width
# justification
#   ^ - center
#   < - left justify
#   > - right justify
# field_width - total number of spaces

# lets say we want to do the following
# print name centered in a field width of 10 with fill symbol *
# print stu_id left justified in a field width of 6 with fill symbol =
# print average right justified in a field width of 7 with 2 numbers after decimal wi
```

```
th fill symbol #
# print grade  right justified in a field width of 3 with fill symbol @

formatted_string = '{0:*^10}{1:=<6}{2:#>7.2f}{3:@>5}'.format(name,stu_id,average,grad
e)
print(formatted_string)
```

```
# f strings: new and improved way to format strings in python available in version 3.
6 and up
formatted_string = f'{name:*^10}{stu_id:=<6}{average:#>7.2f}{grade:@>5}'
print(formatted_string)
```

▼ jan19 lecture

algorithm analysis — think about difference in performance between various algorithm solutions to the same problem

algorithm — step by step procedure for solving a certain problem

program — implementation of an algorithm in a programming language

- without an algorithm there can be no program
- program is a realization of an algorithm

why do we analyze algorithms?

- programs consume resources
- algorithms require time for execution and space to store the data to be processed
- analysis pertains to;
  - execution time (time complexity)
  - memory use (space complexity)
- readability does not affect performance of an algorithm

benchmark analysis — tracking the actual time required by the program to compute its result

OBJECTS AND CLASSES

▼ dice class example:

```
import random

class Die:
  def __init__(self, sides):
    self.sides = sides
  def roll(self):
    return random.randint(1, self.sides)

d6 = Die(6)
type(d6) # Die
d6.roll()
d6.roll()
d6.roll()

d20 = Die(20)
d20.roll()
d20.roll()
d20.roll()

for die in [d6, d20]:
  print(die.sides)
  print(die.roll())
```

▼ fibonacci sequence:

specific sequence of numbers seen in nature all the time

divide almost any fib number by the one before it and you get the same numer;

1.618 —> phi (golden ratio)

the golden rectangle — rectangle whose side lengths are successive fib numbers

can be split into squares of length fib number

draw curve from one corner to other and get that famous spiral

▼ jan21 lecture

benchmarking usefulness;

computes the actual time to execute the program (good)

dependent on a particular machine, programming language ... (bad)

alternate method — use characteristics to compare algorithms that are independent of the program and the machine (better way)

performance of an algorithm;

determined by the number of basic operations needed to solve the problem

determined by the size of the problem

typically the number of data points n

Big O Notation — quantify the number of operations or steps that the algorithm will require

execution time can be expressed as the number of steps —> T(n)

for larger values of n, constant 1 is insignificant

order of magnitude is O(n) — describes the part of T(n) that increases the fastest as the value of n increases

often called Big O notation and is expressed as O(f(n)), where f(n) represents the dominant part of the function

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

in other words, T(n) is the exact number of steps and O(n) is the dominant component of T(n) for an algorithm

▼ jan24 lecture

recursion — function that calls itself

cache — avoids recalculating numbers that were already calculated

fibonacci algorithms:

f1 O(n) iterative bottom up

f2 O(2^n) recursive

not good because too slow

f3 O(n) recursion with cache

not good because we run out of memory (recursive depth)

dynamic programming

f4 O(log2(n)) golden ratio equation

not good because it has error due to floating point storage

divide and conquer

mantissa problem — problems with precision when storing long floating point numbers

▼ jan26 lecture

more fibonacci algorithms:

f5 O(log2(n)) matrix multiplication

choose this one

divide and conquer

divide and conquer — divide problem in half where both parts are the same problem

algorithm techniques;

dynamic programming — use a table to store intermediate results to avoid repeat computation

sub problems not identical

ex: using cache

no more repeat computation

sacrifice memory for time

divided and conquer — decompose problem into two or more smaller problems and solve them separately and combine solutions of smaller to give solution

sub problems identical

ABSTRACT DATA TYPES

▼ abstract data types:

ADT — type defined in terms of its data items and associated operations, not its implementation

the class supports the ADT construct within programming languages

- encapsulation:

  object oriented programming — hiding details that don't need to be revealed in order to use the implementation

  interface — methods of an object

    details are hidden —> encapsulation

  encapsulation;

    you don't need to know the details, you just need understand the interface to use the object

    protects object from any unintentional corruption

  encapsulation example — gearbox

    don't need to know how the car gears work, just need to know how to switch gears to to what you want the car to do

    driver communicates via the interface (gear shift)

    different interfaces between automatic and manual cars

- stacks:

  stack — elements added and removed from the same end

    "last in first out"

    push —> add data

    pop —> remove data

- jan28 lecture

greedy algorithm — sometimes works well for optimization problems

- works in steps
  - at each step...
    - take the best one can get right now, without regarding the eventual optimization
    - hope that by choosing a local optimum at each step, one will end up at the global optimum

optimization problem — one wants to find, not just a solution, but the best solution

ways to create a list:

- given an object, the way you initialize it can have an impact on the running time of your program
- concatenation — least favourite, requires a lot of time

```
...
    myList = myList + [i]
```

- appending — better than concatenation

```
...
    myList.append(i)
```

- comprehension — fastest than appending

```
...
  myList = [i for i in range(n)]
```

- list range — fastest

```
...
  myList = list(range(n))
```

type —- determines the possible values for that variable, the operations that can be done on it, and the meaning of the data

- the modification to a program commonly requires a change in one or more of it's data structures

isolation — separating the implementation of a data structure and its behaviour from the implementation of the program that uses this data structure

- minimizes the impact of a change
- you don't want a change in a data structure to require rewriting every procedure that uses the changed structure

abstraction — hide the details of the implementation and only show the possible operations (interface)

- encapsulation enables abstraction

abstract data types — a set of objects sharing the same properties and behaviours

- the properties are its data (representing the internal state of each object)
- the behaviours are its operations or functions (operations on each instance)
- couples its data and operations
- independent of a programming language (it is a formal description)
- object oriented programming emphasizes data abstraction
- promotes design by contract

▼ jan 31 lecture

from formal ADT to implementation

linear structure — a non-indexed container object that can grow (check this statement on the slides)

- ex: lists stacks queues deque pile
- items are stored sequentially

stacks — (ADT) collection with access only to the last element inserted

- last in first out
- example of an ADT design of a stack
  - add element — push()
  - remove element — pop()
  - check top item — peek()
  - is empty — isEmpty()
  - check size —size()
  - empty stack — reset()
- can be used to reverse a sequence due to LIFO
  - by pushing elements of a sequence in a stack, popping them afterwards from the stack would reverse the sequence
- ex: navigating the web, matching parentheses

reversing a sequence

▼ feb2 lecture

infix, prefix, and postfix expressions

- infix — operator appears between variables
  - parentheses required to remove ambiguity
- prefix — operators before operands
  - parentheses not required to remove ambiguity

postfix — operators after operands

- parentheses not required to remove ambiguity

conversion from infix to prefix and postfix expressions;

- transform to fully parenthesized then convert
- operands don’t move, only operators change position

advantages of postfix notation;

- also called reverse polish notation
- it is unambiguous and does not require parenthesis or brackets
- can be used for arithmetic, logic, and algebra
- used for many implementation

general infix to postfix conversion using stack

- split string into list, and analyze and convert the expression from the list going left to right one single element (token) at a time
- ...
- if token is an operand, append it to the end of an output list
- if the token is an opening parenthesis, push it to the stack
- if the token is a closing parenthesis, pop elements from the stack until the corresponding opening parenthesis and append each operator to the end of the output list
- if the token is an operator, check the top of the stack, as long as there are operators with higher or equal precedence than the token, op them out and append them to the output list, then push the token to the stack
- at the end empty the stack into the output

STACKS

- call stack:

  call stack (runtime stack) — last in first out

    primary purpose of the call stack is to store the return address of the caller of a method to be executed (shows which method is being executed and which methods called it)

    store stack frames (call frames)

      contains return address, parameters, local variables

    relation to debugging; can see sequence of method calls and the variables in the call frame

- exceptions

  metaphor — welder

    try:

      weld pipe

    except melted metal drips:

      wipe it out

    except electrode breaks:

      change new electrode

    if don’t know how to handle problem, send to another person

- handling exceptions:

  try-accept blocks;

```
# exception will run if either line n try block gets an error
try:
  file = open('testfile.txt')
  var = bad_var
except Exception:
  print('Sorry. This file does not exist')

# exception will only run if file open line in try block gets an error
try:
```

```
  file = open('testfile.txt')
  var = bad_var
except FileNotFoundError:
  print('Sorry. This file does not exist')

# first exception will only run if file open line in try block gets an error
# second exception will only run if file line runs successfully but var line gets an
 error
try:
  file = open('testfile.txt')
  var = bad_var
except FileNotFoundError:
  print('Sorry. This file does not exist')
except Exception:
  print('Sorry something went wrong')

# if exception runs, will print out just the error it ran into (no traceback)
try:
  file = open('testfile.txt')
  var = bad_var
except Exception as e:
  print(e)
```

else-finally blocks;

```
# else block runs if the try block doesn't run the except block
# finally block runs no matter what happens
try:
  f = open('testfile.txt')
  var = bad_var
except Exception as e:
  print(e)
else:
  print(f.read())
  f.close()
finally:
  print('Executing Finally...')
```

can raise exceptions on your own... i.e. error doesn’t have to be a language specified error

```
# example
try:
  f = open('curruptfile.txt')
```

```
  if f.name == 'curruptfile.txt':
    raise Exception
except Exception as e:
  print(e)
else:
  print(f.read())
  f.close()
finally:
  print('Executing Finally...')
```

▼ feb4 lecture

infix to postfix

```
for token in tokenList:
  # if the token is an operand
  if token in alphaOperand or token in digitOperand:
    postfixList.append(token)
  elif token == '(':
    opStack.push(token)
  elif token == ')':
    topToken = opStack.pop()
    while topToken != '(':
      postfixList.append(topToken)
      topToken = opStack.pop()
  else:
    # if the token is not an operand
    while (not opStack.isEmpty()) and (prec[opStack.peek()] >= prec[token]:
    opStack.push(token)

while not opStack.isEmpty():
  postfixList.append(opStack.pop())
return ' '.join(postfixList)
```

look infix to postfix algorithm handout

stacks can be used;

to reverse a sequence

to navigate the web

traversing a maze

parser matching parenthesis

infix prefix and postfix expressions

maze trace — depth first search of a maze

in each position...

push all legal moves that were not visited

pop one position and move it

fail if stack is empty

success if final

exception — event which occurs during the execution of a program that disrupts the normal flow of the program's instructions

semantic error

exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions

exception is a python object that represents an error

use try and except blocks

once an exception has been handled, processing continues normally on the first line after the try except block

if you don't catch the exception, your entire program will crash

- except *Exception1*: catches Exception 1
- An except clause may name multiple exceptions as a parenthesized tuple, for example:

  except (RuntimeError, TypeError, NameError):
- An except clause without explicit exception will catch all remaining exception

```
try:
  some statements
except: print("Unexpected error:")
```

▼ feb7 lecture

advantages of using exceptions;

separating error-handling code from regular code

deferring decisions about how to respond to exceptions

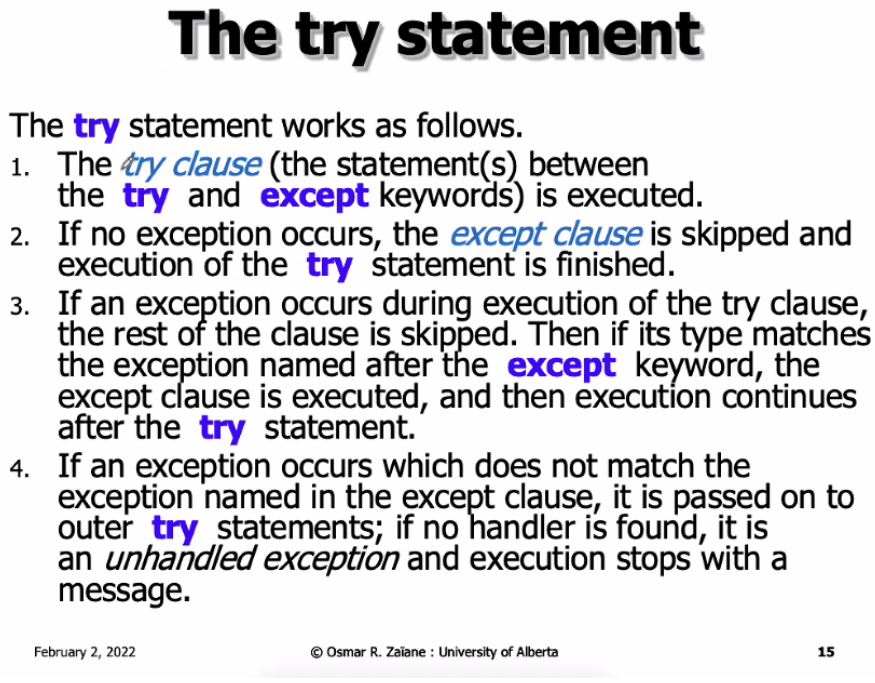
providing a mechanism for specifying the different kinds of exceptions that can arise in our program

try statement — made up of a try and except block

once an exception has been handled, processing continues normally on the first line after the try-except block

may have more than one except clause, however at most one handler will be executed

handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement



if exception not handled, statement after try statement is not done

implicit propagation — if an error is not handled in a function, it is propagated to the function that called the function where the error occurred

- automatically happens

propagating exceptions —

- exception will bubble up the call stack until...
  - it reaches a method with a suitable handler or
  - it propagates through the main program (the first method on the call stack)
- if it is not caught by any method, the exception is treated like an error (stack frames are displayed and the program terminates)

raising exception to caller;

- enables you to defer the handling of an error to the function that called the function that got the error

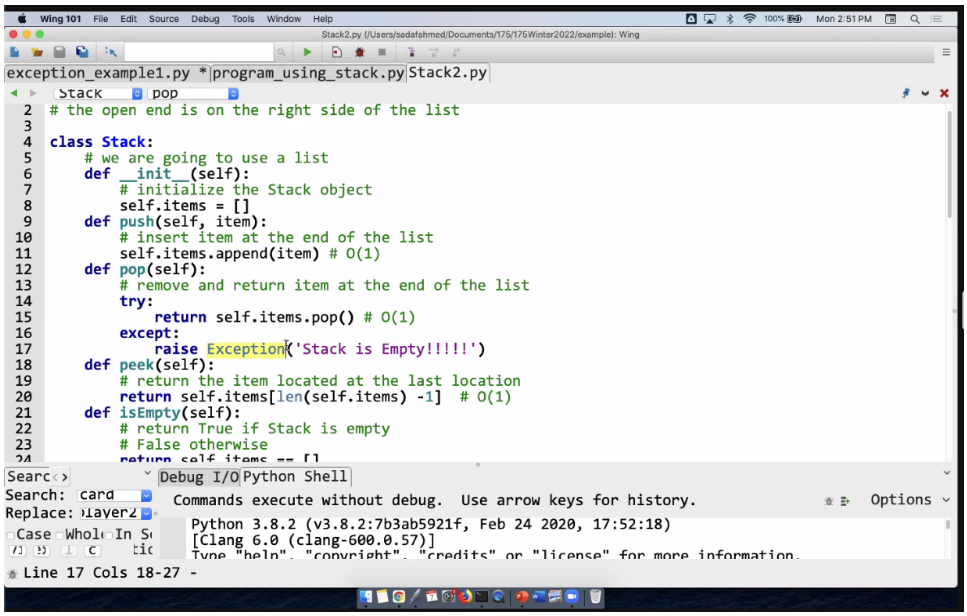
```
def main():
  func()
  try:
    ...
  except:
    ...

def func():
  try:
    ...
  except Exception:
    raise

main()
```

using exception objects:

- exception can have an argument which is a value that gives additional info about the problem
  - the contents of the argument vary by exception

this argument receives the value of the exception mostly containing the cause of the exception

```
try:
  888
except Exception as e:
  print('There was an exception\n', e.args) # args is an attribute of the class e
 (Exception)
```



▼ feb9 lecture

using exception objects;

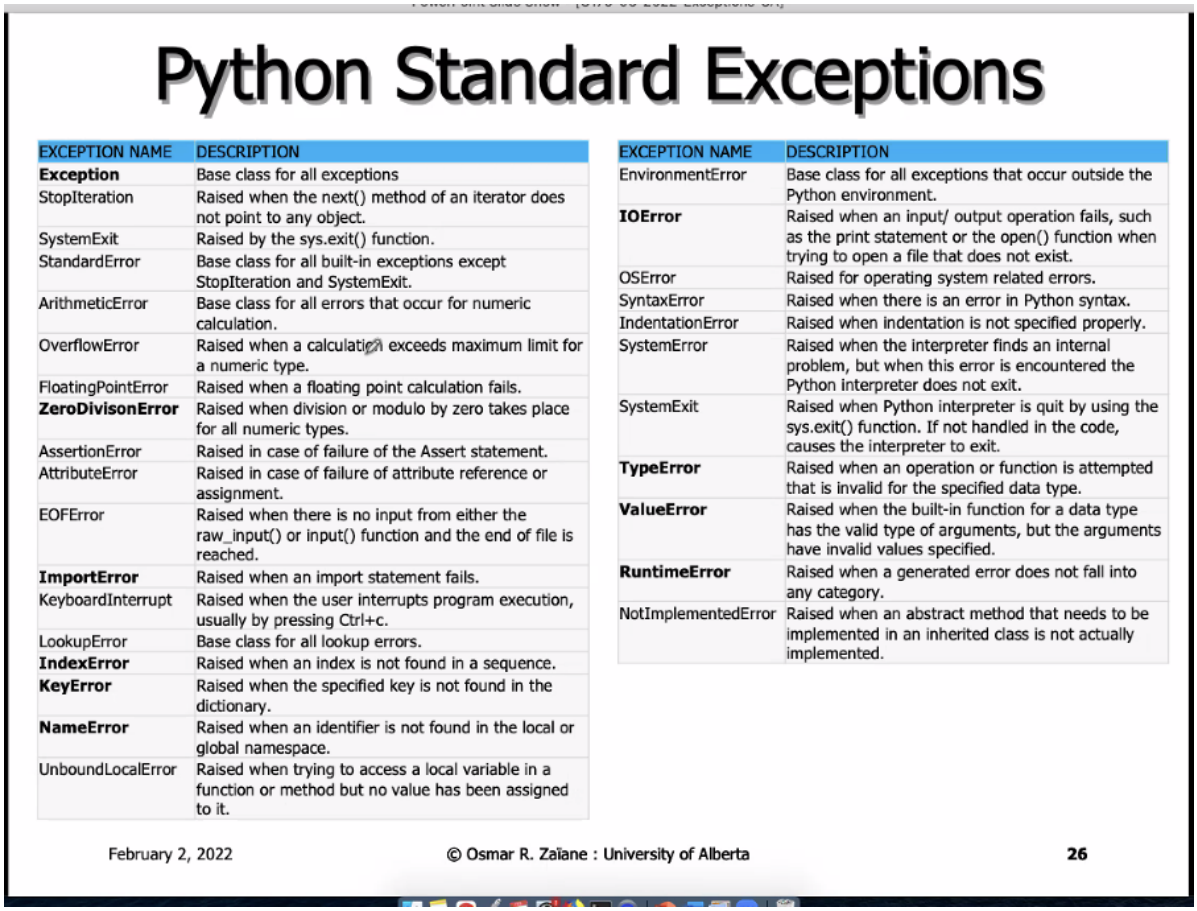
exception can have an argument, which is a value that gives additional info about the problem

the contents of the argument vary by exception

this argument received the value of the exception mostly containing the cause of the exception

by assigning/naming the exception to a variable, you can access the attributes of the object

.args is a tuple



Python Standard Exceptions

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.

EXCEPTION NAME	DESCRIPTION
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

February 2, 2022 © Osmar R. Zaïane : University of Alberta 26

raising exceptions;

- what can be raised as an exception?
  - any standard python exception
  - a new instance of Exception
  - instances of our own specialized exception classes

else clause;

- optional clause in a try except statement
- when present, must follow all except clauses
- runs if the try is successful (no exception is raised)

finally clause;

optional clause in a try except statement

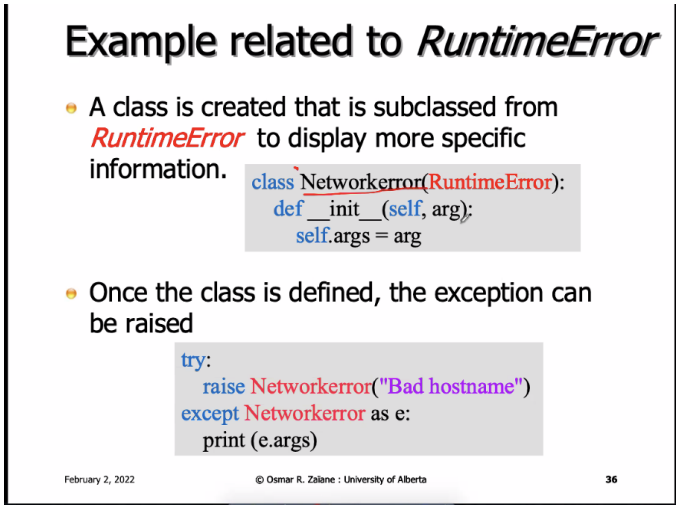
runs no matter what

useful if you want to perform some kind of ‘clean up’ operations before exiting the method (ex: closing a file)

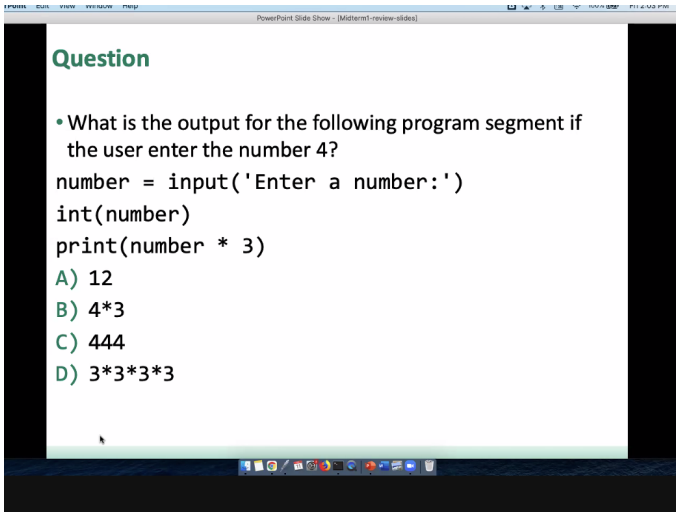
also avoids duplicating code in each except clause

slides 31-33 describes how exceptions are processed (in words) VERY IMPORTANT FOR THE MIDTERM

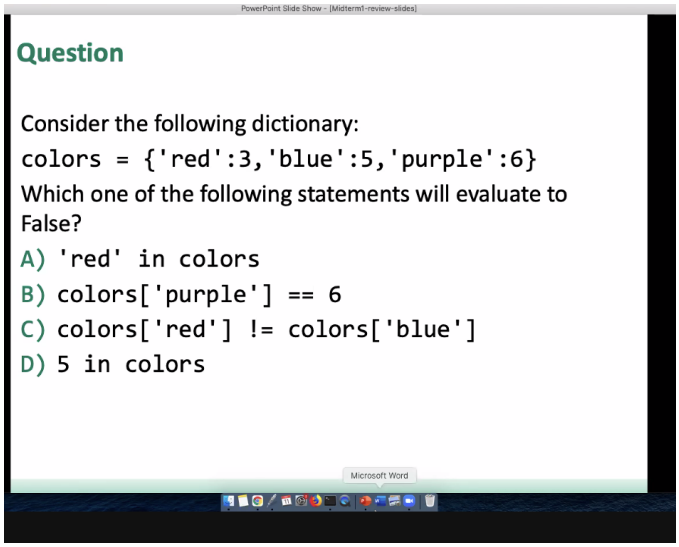
example related to RuntimeError:



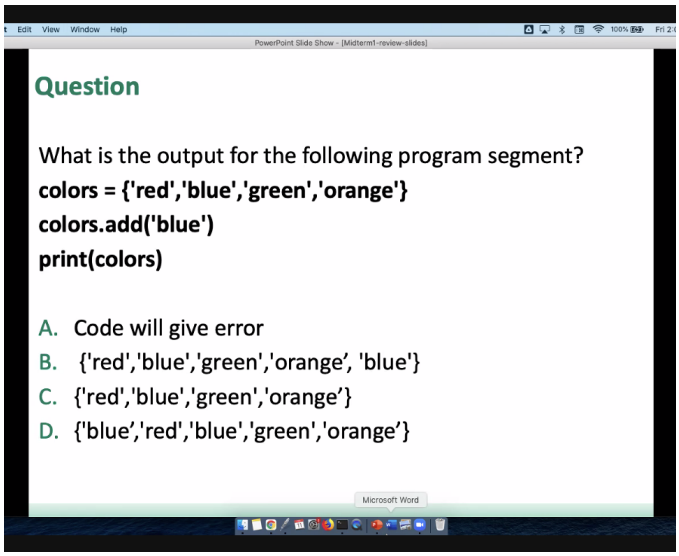
▼ midterm review


PowerPoint Slide Show - [Midterm1-review-slides]
Question
• What is the output for the following program segment if the user enter the number 4?
number = input('Enter a number:')
int(number)
print(number * 3)
A) 12
B) 4*3
C) 444
D) 3*3*3*3

C


PowerPoint Slide Show - [Midterm1-review-slides]
Question
Consider the following dictionary:
colors = {'red':3,'blue':5,'purple':6}
Which one of the following statements will evaluate to False?
A) 'red' in colors
B) colors['purple'] == 6
C) colors['red'] != colors['blue']
D) 5 in colors
Microsoft Word

D



C

▼ feb11 lecture

assertions — statement that raises an AssertionError exception if a condition is not met

```
assert Expression[,Arguments]
```

if the assertion fails, python uses the given argument for the Assertion Error

AssertionError can be caught and handed like any other exception

it is good practice to place assertions at the start of a function to check for valid input and after a function call to check for valid output

queue — ordered collection where items are added at one end (rear/tail) and removed at the other end (front/head)

first in first out (FIFO)

enqueue (add) from rear, dequeue (remove) from front

useful for...

printing queue for waiting printing tasks

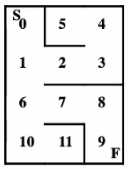
computing processes waiting list

keyboard buffer

traversing maze

maze algorithm queue example;

- Like Stacks, Queues can also be used to store unsearched paths.
- Repeat as long as the current square is not null and is not the finish square:
  - "Visit" the square and mark it as visited.
  - Enqueue one square on the queue for each unvisited legal move from the current square.
  - Dequeue the queue into the current square or bind the current square to null if the queue is empty.
- If the current square is the goal we are successful, otherwise there is no solution



February 9, 2022 © Osmar R. Zaïane : University of Alberta 6

when using stack, he search goes deep along a single path as possible before trying another path

when using queue, search expands a frontier of nodes equidistant from the start (breadth first search)

stack vs queue is not necessarily better than the other... depends on the maze