# CSC148H Week 4

Ilir Dema, Michael Miljanovic

Summer 2021

# ADT vs. Data Structure

- Abstract Data Type (ADT)
  - Tells you about the data it manages and the operations it supports
  - Does *not* tell you how it is implemented
  - It's an interface (like the public methods/attributes of an object)

# ADT vs. Data Structure...

- Data Structure
  - A chosen implementation strategy for an ADT
  - Decides how to actually store the data and implement the operations

# Purpose of ADTs

- ADTs capture common patterns of manipulating data
- Helps us work and communicate at higher levels of abstraction
- Fluency with ADTs is required for any computer scientist or programmer

# Stack Application: Balanced Parentheses

Does a string correctly use matching parentheses, brackets, and braces?

- `(a * b) + c` good
- `a * ) b + (c` bad
- `(a + [b - {c * d}])` good
- `(a + [b - {c * d)]}` bad
- `(ab(cd(e)fg))(h(i(j)(k(l)))m(n))` thoughts?

# Stack Application: Balanced Parentheses...

Why do we care?

- Your IDE checks for well-formed Python code
- HTML: determine whether elements are properly nested

# Defining Balanced

- A string with no parentheses is balanced
- A string that begins with a left parenthesis `(`, ends with a right parenthesis `)`, and is balanced in between is balanced. Same for brackets `[...]` and braces `{...}`
- The concatenation of two strings with balanced parentheses is also balanced: `(...)` `(...)`

# Simplified Problem

To start, let's focus on only parentheses.

- Ignore all characters except ( and )
- Keep track of when you see a ( but forget about it when you've seen the matching )

# Worksheet 1

Worksheet 1, balanced parentheses

# The Full Problem

What do we do when brackets and braces are in the mix?

- Stack can still be used to check whether it is balanced
- Push the opening parentheses/brackets/braces, pop the closing ones

# Exceptions

- So far, our Python functions have returned values or modified objects
- But what if our function cannot complete successfully – then what should happen?
  - e.g. Calling pop on an empty stack
  - e.g. Trying to create a fraction with 0 denominator

# Exceptions...

- In some languages (e.g. C), functions return "special values" to signify errors
  - Lots of Unix functions return `-1` to mean "error"
- Two concerns with that approach
  - It requires you to check the return value of every function you call
  - It assumes there is an appropriate error value to return that won't be confused with a real return value!

# What are Exceptions?

- Exceptions allow you to structure code in a natural way so that error handling and recovery are isolated from the regular flow of your program
- An *exception* is an object that indicates an exceptional situation (not necessarily a problem)
- An exception gets raised during program execution and transfers control to an exception handler
- Exceptions must be "caught" by an exception handler, or your program will crash

# Examples of Exceptions

```
>>> 10 * (1 / 0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + junk
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'junk' is not defined
>>> junk = 'abc'
>>> 4 + junk # can't add int and str
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: ...
```

# Raising Exceptions

Two forms:

```
>>> raise ValueError
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError
>>> raise ValueError('invalid time/date value')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid time/date value
```

# User-Defined Exceptions

You can make new types of exceptions (inherit from `Exception`):

```
>>> class EmptyStackError(Exception):
...     pass
...
>>> raise EmptyStackError('pop from empty stack')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.EmptyStackError: pop from empty stack
```

# Handling Exceptions

- If a piece of code can raise an exception, you can put it in a `try` block
- If there is an associated `except` block that matches the kind of exception raised, then that block will handle the exception
- There may be more than one `except` block that matches; in this case, the first is used
- An `except` block "matches" a raised exception if the `except` block names the same class or a superclass of the exception

# Example: Handling Exceptions

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s)
    print('success')
except IOError:
    print('Input/Output error.')
except ValueError:
    print('Could not convert data to an integer.')
print('continuing')
```

# Execution with Exceptions

- If no exception occurs, no `except` block is executed
- If an exception occurs and an `except` block handles it, execution continues following the enclosing `try` block
- If an exception occurs and no `except` block handles it, the exception propagates up the function call stack until it is handled or terminates the program

# Worksheet 2

Worksheet 2, Stack Size

# ADT Puzzle

You're given a list of integers; your goal is to transform the list into a new list according to the following rule:
Find the leftmost pair of consecutive numbers in the list whose values are $x$ and $x + 1$, replace them by the single element whose value is $2x + 1$ and repeat the process using this new list. If no pair of integers satisfies this property, the process is complete.

- Example: list $[1, 2, 3, 4]$ is transformed first to $[3, 3, 4]$, and then to $[3, 7]$

# ADT Puzzle...

- What is the problem with using the "obvious" algorithm of scanning left to right looking for the next pair of numbers satisfying the condition?
- Example: $[32, 16, 8, 4, 2, 1, 2]$
- Which ADT can we use to speed things up?

# Worksheet 3

Worksheet 3, Efficiency

# Test 1 Info

See the announcement.