# How to Handle Exceptions

- To capture an exception so that the program won't terminate:

- ***try***
  …
  ***except*** _____***:***
  …

- where … are statements to be executed and _____ is an exception clause

- An ***else*** clause can be added to indicate code that should run if no exception occurred in the ***try*** clause

- A ***finally*** clause can be added to indicate code that must always be run, whether or not an exception occurred in the ***try*** clause

# *try/except* Example

```
#!/usr/bin/python3.11

try:
    a = float(input('Enter a:\n'))
    b = float(input('Enter b:\n'))
    if b == 0:
        raise Exception('Cannot Divide by 0')
    c = a / b
except ValueError as details:
    print(str(details))          # Called if float fails
except Exception as details:
    print(str(details))          # Called if 0 entered
else:
    print('a / b = ' + str(c))   # Called if no issue occurred
finally:
    print('Done')                # Always called
```

- ***float(s)*** converts *s* to a floating point number

# Exercises

- Modify the prime checker so that it prints out an error message if the input is not a number. Use a ***try***, not an ***if*** statement, to detect this scenario.

- Furthermore, print an error message if num < 2; be sure to use ***raise*** for this.

# File I/O

- Python supports reading from and writing to files
- Traditionally, this followed the pattern ***open***, ***read*** or ***write***, and ***close***
- Closing files is easy to forget (potentially crashing the app when there are too many open file handles) and hard to get right (when handling exceptions; when closing too quickly after a ***write***)
- A better approach is use of the ***with*** statement

# File I/O Example

```
#!/usr/bin/python3.11

def write_temp(temp: str, file_name: str) -> None:
    with open(file_name, 'w') as f:
        f.write(str(temp))


def read_temp(file_name: str) -> str:
    with open(file_name, 'r') as f:
        return f.read()


try:
    file = '/tmp/data'
    write_temp('Hello World', file)
    print(read_temp(file))
except OSError as details:
    print('Error', details)
```

# Another File I/O Example

```
#!/usr/bin/python3.11

file = '/tmp/data'

# Read and print out the given file line by line
try:
   with open(file) as f:
      for line in f:
         print(line)
except OSError as details:
   print('Error', details)
```

- If we don't need to do anything other than catch the exception, we can also use:

```
except OSError:
   pass
```

# How to Call a Function

- Functions in Python are similar to methods in Java
- Say we have a function

```
def add_values(val1, val2):
    return val1 + val2
```

- This function adds the two parameters that are passed in and then returns the result
- Say we have two variables, a = 7 and b = 11. To add *a* and *b* using the *add* function and saving the result in variable *c*:
  ***c = add_values(a, b)***

# How to Call a Function

- It is possible to add type annotations and comments

```
def add_values(val1: int, val2: int) -> int:
    """
    Computes and returns val1 + val2.

    :param val1: The first value to be added
    :param val2: The second value to be added
    :return: val1 + val2
    """
    return val1 + val2
```

- While Python makes this optional, it is good software engineering practice to use these language features!

# Default Arguments

- It is possible to add type annotations and default values

```
def confirm(prompt: str = "OK?") -> bool:
    return input(prompt + ': ').lower() in ('y', 'yes')
```

- Can then avoid providing a second argument

```
print(confirm('Really delete?'))  # uses 'Really delete?'
print(confirm())                  # uses 'OK?'
```

# Exercises

- Write a function that takes in a number and returns *True* if it is prime, *False* otherwise

# Scope

```
reply = input('Install? ')
if reply.lower() in ('y', 'yes'):
    install = True

print(install)
```

- *install* is defined in the *if* statement
- Unlike Java, should *if* evaluate to False, *install* will be undefined

# Scope

```
def prompt() -> bool:
    install = False
    reply = input('Install? ')
    if reply.lower() in ('y', 'yes'):
        install = True
    return install


prompt()
print(install)
```

- *install* is defined in the *prompt* function, so *install*'s scope is limited to *prompt*
- Once execution leaves the *prompt* statement, *install* is no longer defined

# Scope

```
def prompt1() -> bool:
    install = False
    reply = input('Install? ')
    if reply.lower() in ('y', 'yes'):
        install = True
    return install


def prompt2() -> bool:
    install = False
    reply = input('Install? ')
    if reply.lower() in ('y', 'yes'):
        install = True
    return install
```

- *install* in *prompt1* and *install* in *prompt2* are two ***separate*** variables

- They look the same, but are different!

# Scope

```
install = False


def prompt() -> bool:
    reply = input('Install? ')
    if reply.lower() in ('y', 'yes'):
        install = True
    return install


prompt()
print(install)
```

- *install* in *prompt* and *install* in the global scope are **not** the same

# Scope

```
install = False


def prompt() -> bool:
    global install
    reply = input('Install? ')
    if reply.lower() in ('y', 'yes'):
        install = True
    return install
```

- *install* in *prompt* and *install* in the global scope **are** the same
- ***global*** is best avoided

# Lists

- Say you are asked to write a program that manages a grocery list
- How can you keep track of every item on the list?
- Tedious:
  x = 'bananas'
  y = 'apples'
  z = 'oranges'
- Easier:
  items = ['bananas', 'apples', 'oranges']
- Python supports just that!

# List Examples

```
#!/usr/bin/python3.11

shopping_list = ['bananas', 'apples', 'oranges']
for item in shopping_list:
    print('Do not forget the ' + item)
```

- Or:

```
#!/usr/bin/python3.11

shopping_list = ['bananas', 'apples', 'oranges']
num_items = len(shopping_list)
for i in range(0, num_items):
    print('Do not forget the ' + shopping_list[i])
```

# Example With Parameter Passing

```
#!/usr/bin/python3.11

queue = []


def add_to_queue(item: any, q: list) -> None:
    q.append(item)


def remove_from_queue(q: list) -> any:
    if len(q) > 0:
        # item = q[0]
        # q.remove(item)
        # return item
        return q.pop(0)
    else:
        return None
```

# Example With Parameter Passing

```
command = ''
while True:
    command = input('(a)dd (r)emove (q)uit?\n')
    match command:
        case 'q' | 'Q':
            break
        case 'a' | 'A':
            val = input('Item to add?\n')
            add_to_queue(val, queue)
        case 'r' | 'R':
            val = remove_from_queue(queue)
            print('Removed', val)
        case _:
            print('Unknown command')

    for val in queue:
        print('>    ' + str(val))
```

# Other List Commands

- Assuming *lst* is a list:
  - ***lst.insert(i, x)*** -- inserts *x* at index *i*; if *i* is too large, will append; if *i* is negative, will prepend
  - ***lst.count(x)*** -- number of times that *x* occurs in the list
  - ***lst.index(x)*** -- the index at which *x* first occurs in the list; will throw a *ValueError* if not found
  - ***lst.reverse()*** -- reverses a list in place
  - ***lst.sort()*** -- sorts a list in place
  - ***lst.copy()*** or ***lst[:]*** -- returns a shallow copy of the list
  - ***lst.clear()*** or ***del x[:]*** -- removes all elements from the list

# Lambda Functions

- It is possible to create anonymous (nameless) functions in Python

```
def example(x):
    return lambda y: y + x # returns a function with 1 parameter

e = example(10) # e(x) returns 10 + x
print(e(20), e(30))  # prints 30 40
```

- The above *lambda* uses the *x* from the *example* parameter list to set up an anonymous function that:
  - takes in a value *y*,
  - adds *x* to it, and then
  - returns the sum of *x* and *y*

# Applying an Operation to All List Elements

- Let's say we want to double all numbers in a list
- Instead of:

```
lst = [1, 2, 3, 4]
new_lst = []
for i in lst:
    new_lst.append(i * 2)
print(new_lst) # prints [2, 4, 6, 8]
```

# Lambdas, Maps, and Lists

- We can use:

```
lst = [1, 2, 3, 4]
new_lst = list(map(lambda i: i * 2, lst))
print(new_lst) # prints [2, 4, 6, 8]
```

- The ***map*** function applies the *lambda* to every element in *lst*
- The list constructs a new list, using the result of the ***map*** function

# List Comprehensions

- We can also use:

```
lst = [1, 2, 3, 4]
new_lst = [i * 2 for i in lst]
print(new_lst) # prints [2, 4, 6, 8]
```

- The ***for*** steps through every element in *lst*
- Each element is temporarily assigned the value *i*, which is then used in the computation; the result is appended to a new list

# List Comprehensions

- Can add further ***if*** (and ***for***) statements:

```
lst = [1, 2, 3, 4, 'hello']
new_lst = [i * 2 for i in lst if type(i) == int and i > 0]
print(new_lst) # prints [2, 4, 6, 8]
```

- Here, we only apply the operation to positive integers; the rest is skipped

# More Examples

- Given
  ***x = ['10', '20', '30', '40']***

- ***[val for val in x if int(val) < 25]***
  results in
  *['10', '20']*

- ***[y.isnumeric() for y in x]***
  returns
  *[True, True, True, True]*

- ***any(y.isnumeric() for y in x)***
  returns
  *True*

# Exercises

- Write a function that takes a list of integers and returns a list of integers that has only out of order numbers, assuming strictly descending order. For example, if the list is *[7, 7, 4, 6, 4, 2, 0]*, the result is *[7, 6, 4]* because the second 7, the 6, and the second 4 are not strictly smaller than the previous numbers

- Write a function that takes a list of integers and returns a list of strings of asterisks, where the *len(asterisk_list[i])* == *integer_list[i]*. For example, *[1, 2, 3]* would result in *[*, **, ***]*. Avoid using comprehensions. Ignore non-negative integers.

- Repeat the previous question using comprehensions.

# Multidimensional Lists

- a = ['Hello'] * 2 creates a list with two 'Hello' elements inside
- b = ['Hello' for _ in range(2)] also creates a list with two 'Hello' elements inside
- However, these two approaches are quite different...

# * Notation for Lists

- board = [['] * 2] * 2
- Result:
  " "
  " "
- board[0][1] = 'X'
- Result:
  " 'X'
  " 'X'
- The * resulted in a shallow copy, so the two rows of the list are the same!

# for _ in Notation for Lists

- board = [[" for _ in range(2)] for _ in range(2)]
- Result:
  " "
  " "
- board[0][1] = 'X'
- Result:
  " 'X'
  " "
- The ***for*** creates 2 entirely new lists

# Exercises

- Write a function that takes in an integer *n* and returns an *n* x *n* list where the two diagonals contain asterisks, and all the other entries contain hyphens. For example, if *n* is 4, the result is *[['*', '-', '-', '*'], ['-', '*', '*', '-'], ['-', '*', '*', '-'], ['*', '-', '-', '*']]*, i.e.,

```
'*'  '-'  '-'  '*'
'-'  '*'  '*'  '-'
'-'  '*'  '*'  '-'
'*'  '-'  '-'  '*'
```