COL1000: Introduction to Programming

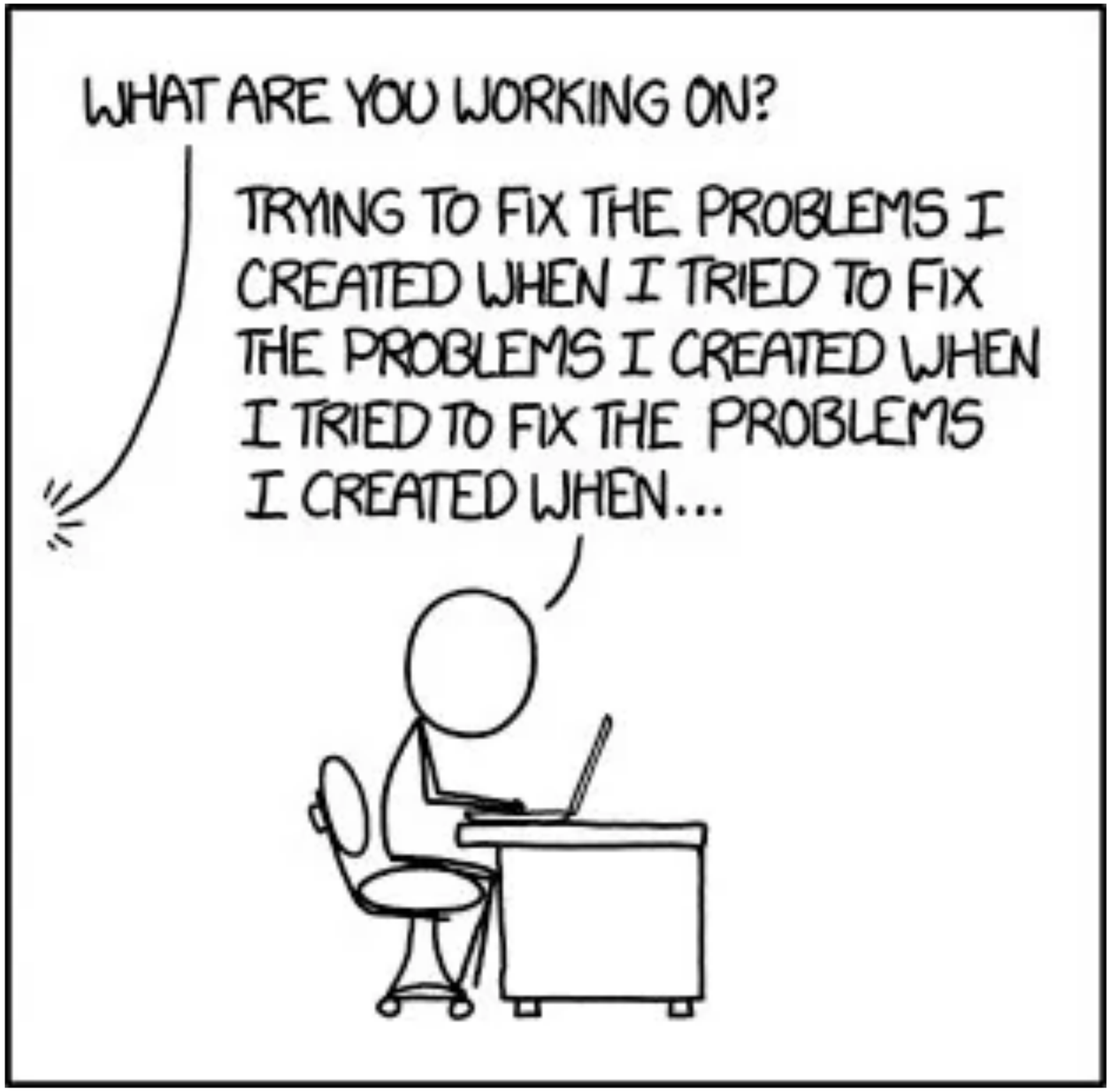
Exceptions, Specifications

Subodh Sharma | Lec 25 | Oct 15

RECAP

- Manual Memoization
- Decorators
- Today — Exceptions
- Today — Specifications & Debugging

Exceptions


WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...

Functions: Exceptions

Functions: Exceptions

- Exceptions in functions **denote errors or unexpected conditions**

Functions: Exceptions

- Exceptions in functions **denote errors or unexpected conditions**
- There are two ways of handling exceptions in the code:

Functions: Exceptions

- Exceptions in functions **denote errors or unexpected conditions**
- There are two ways of handling exceptions in the code:
  - Explicitly **raising the errors with an appropriated exception message**

Functions: Exceptions

- Exceptions in functions **denote errors or unexpected conditions**
- There are two ways of handling exceptions in the code:
  - Explicitly **raising the errors with an appropriated exception message**
  - **Catching** them to handle them gracefully

Functions: Exceptions

- Exceptions in functions **denote errors or unexpected conditions**
- There are two ways of handling exceptions in the code:
  - Explicitly **raising the errors with an appropriated exception message**
  - **Catching** them to handle them gracefully
- There are **built-in** exceptions and **customised** exceptions

Functions: Exceptions

- Exceptions in functions **denote errors or unexpected conditions**
- There are two ways of handling exceptions in the code:
  - Explicitly **raising the errors with an appropriated exception message**
  - **Catching** them to handle them gracefully
- There are **built-in** exceptions and **customised** exceptions
  - In this course, we will deal with only built-in exceptions!

Functions: Exceptions Hierarchy

```
BaseException
├─ SystemExit
├─ KeyboardInterrupt
├─ GeneratorExit
└─ Exception
   ├─ ArithmeticError
   │  ├─ ZeroDivisionError
   │  ├─ OverflowError
   │  └─ FloatingPointError
   ├─ AssertionError
   ├─ AttributeError
   ├─ BufferError
   ├─ EOFError
   ├─ ImportError
   │  └─ ModuleNotFoundError
   ├─ LookupError
   │  ├─ IndexError
   │  └─ KeyError
   ├─ NameError
   │  └─ UnboundLocalError
   ├─ OSError
   │  ├─ FileNotFoundError
   │  ├─ PermissionError
   │  ├─ BlockingIOError
   │  ├─ TimeoutError
   │  ├─ IsADirectoryError
   │  ├─ NotADirectoryError
   │  └─ ConnectionError
   │     ├─ BrokenPipeError
   │     ├─ ConnectionAbortedError
   │     ├─ ConnectionRefusedError
   │     └─ ConnectionResetError
   ├─ ReferenceError
   ├─ RuntimeError
   │  ├─ NotImplementedError
   │  └─ RecursionError
   ├─ StopIteration
   ├─ StopAsyncIteration
   ├─ SyntaxError
   │  └─ IndentationError
   ├─ TypeError
   ├─ ValueError
   │  └─ UnicodeError
   │     ├─ UnicodeDecodeError
   │     ├─ UnicodeEncodeError
   │     └─ UnicodeTranslateError
   └─ Warning (actually separate hierarchy under warnings module)
```

Functions: Exceptions

Functions: Exceptions

- **Raising Exceptions**

Functions: Exceptions

- **Raising Exceptions**
  - `raise` creates an exception object and changes the control-flow

Functions: Exceptions

- **Raising Exceptions**
  - `raise` creates an exception object and changes the control-flow
  - Python runtime catches the error and matches with the right exception (if handler is provided, else produces a traceback)

Functions: Exceptions

- **Raising Exceptions**
  - `raise` creates an exception object and changes the control-flow
  - Python runtime catches the error and matches with the right exception (if handler is provided, else produces a traceback)

```
def Div(x: float, y: float) -> float:
    if not isinstance(a, (int, float)) or \
       not isinstance(b, (int, float)):
        raise TypeError("division requires x and y to be numbers")
    if b == 0:
        raise ZeroDivisionError()
```

Functions: Exceptions

- **Raising Exceptions**
  - `raise` creates an exception object and changes the control-flow
  - Python runtime catches the error and matches with the right exception (if handler is provided, else produces a traceback)

```
def Div(x: float, y: float) -> float:
    if not isinstance(a, (int, float)) or \
       not isinstance(b, (int, float)):
        raise TypeError("division requires x and y to be numbers")
    if b == 0:
        raise ZeroDivisionError()
```

Functions: Exceptions

- **Raising Exceptions (try-except block)**
  - creates an exception object, catches the exception
  - Python runtime goes through the call stack looking for a matching `except` **code**
    - **Control-flow resumes**

Exception Handling: Raise vs Try-Except

- **Raise** could be used for enforcing preconditions, postconditions or assert invariants
- **Try-except**: If you want to have graceful recovery from an error, log the error
  - The execution can **potentially continue** after the `except` block
- What is raised exception is not caught?
  - Traceback is shown by the Python runtime

Exception Handling: Raise vs Try-Except

- **Raise** could be used for enforcing preconditions, postconditions or assert invariants
- **Try-except**: If you want to have graceful recovery from an error, log the error
  - The execution can **potentially continue** after the `except` block

```
try:
    x = int(input("Enter an integer: "))
    print(10 / x)
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("Division by zero is undefined.")

print("continuing the execution")
```

```
bash-3.2$ python3 ./lec24.py
Enter an integer: 0
Division by zero is undefined.
continuing the execution
```

Exception Handling: Raise vs Try-Except

- **Raise** could be used for enforcing preconditions, postconditions or assert invariants
- **Try-except**: If you want to have graceful recovery from an error, log the error
  - The execution can **potentially continue** after the `except` block

```
try:
    x = int(input("Enter an integer: "))
    print(10 / x)
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("Division by zero is undefined.")

print("continuing the execution")
```

```
bash-3.2$ python3 ./lec24.py
Enter an integer: 0
Division by zero is undefined.
continuing the execution
```

Continuing the execution after catching the exception

Exception Handling: Try-Except

- **Try-except**: If you want to have graceful recovery from an error, log the error
  - The execution can **potentially continue** after the `except` block
  - Can combine several exception types into group and treat them similarly.

```
try:
    x = int(input("Enter an integer: "))
    print(10 / x)
except (TypeError, ZeroDivisionError):
    print("Error: Either invalid input or Division by zero.")

print("continuing the execution")
```

Exception Handling: Try-Except

- **Try-except**: If you want to have graceful recovery from an error, log the error
  - The execution can **potentially continue** after the `except` block
  - Can combine several exception types into group and treat them similarly.

```
try:
    x = int(input("Enter an integer: "))
    print(10 / x)
except (TypeError, ZeroDivisionError):
    print("Error: Either invalid input or Division by zero.")

print("continuing the execution")
```

Grouping the exceptions

Exception Handling: Else and Finally Clauses

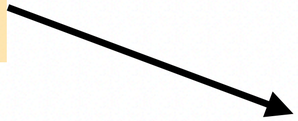
- `try-except` block can be extended with `else` and `finally` clauses to provide more control
  - `Else:` This block is executed **only if no** exceptions are raised in the `try` block

```
try:
    x = int(input("Enter an integer: "))
    print(10 / x)
except (TypeError, ZeroDivisionError):
    print("Error: Either invalid input or Division by zero.")
else:
    print("great no errors were caught")

print("continuing the execution")
```

Exception Handling: Else and Finally Clauses

- `try-except` block can be extended with `else` and `finally` clauses to provide more control
  - `Else:` This block is executed **only if no** exceptions are raised in the `try` block

```
try:
    x = int(input("Enter an integer: "))
    print(10 / x)
except (TypeError, ZeroDivisionError):
    print("Error: Either invalid input or Division by zero.")
else:
    print("great no errors were caught")

print("continuing the execution")
```



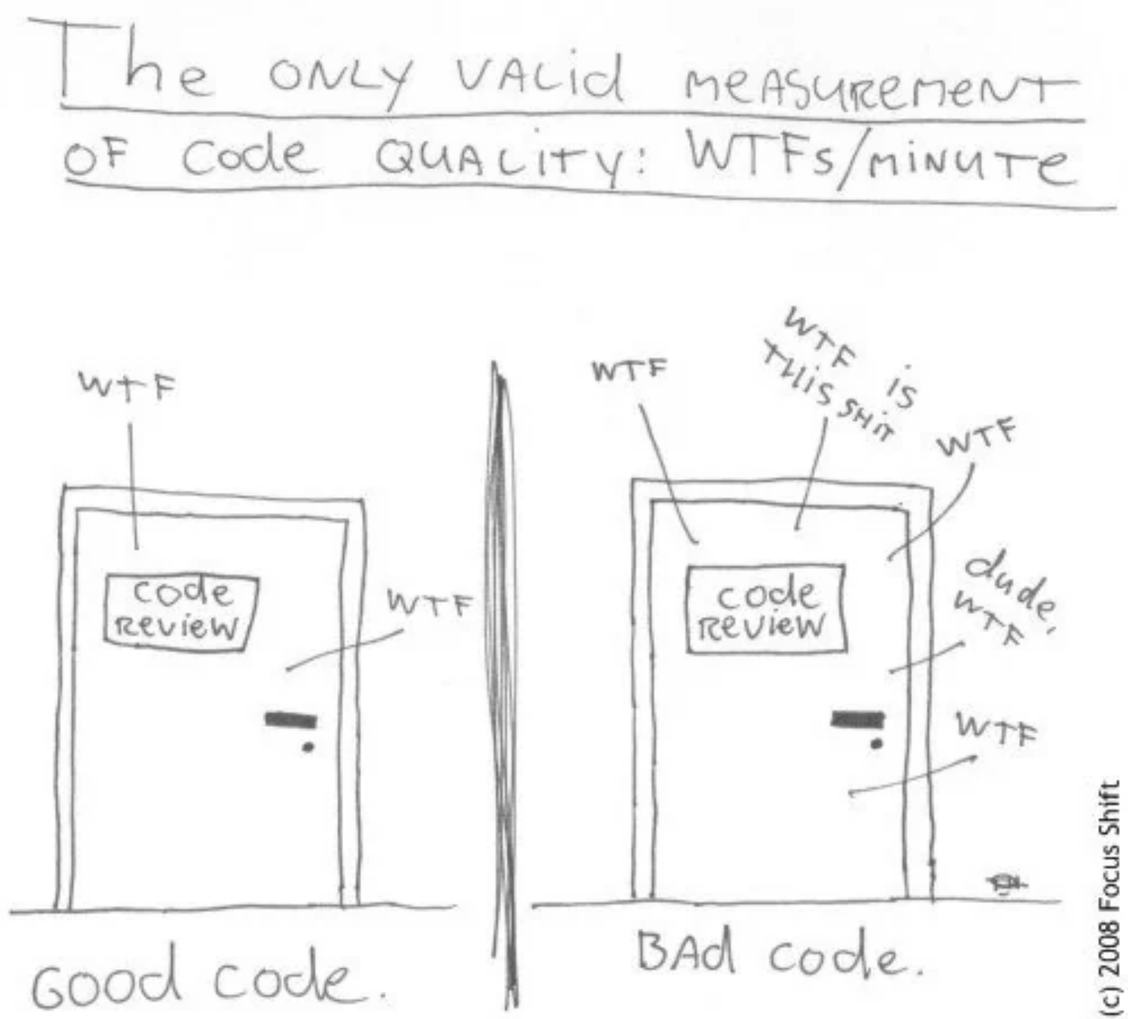
Executed if no errors were caught

Exception Handling: Else and Finally Clauses

- `try-except` block can be extended with `else` and `finally` clauses to provide more control
  - `finally:` This block **always execute —** perfect for clean up code!
    - **Runs before execution resumes after `try-except-else-finally` block**

```
def Div() -> float:
    try:
        x = int(input("Enter x: "))
        y = int(input("Enter y: "))
        if not isinstance(x, int) or \
           not isinstance(y, int):
            raise TypeError("Type should either be int or float")
        if y == 0:
            raise ZeroDivisionError("Divisor cannot be 0")
        #return x/y
    except TypeError as e:
        print(f"Problem Occured: {e!r}")
    except ZeroDivisionError as e:
        print(f"Problem Occured:{e!r}")
    else:
        print("no errors found, returning the output")
        return x/y
    finally:
        print("closing the program execution")

Div()
```

Specifications



Specifications

Specifications

- States what the code **must** do (and not how).

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**
    - **Good**: **let n = len(s), r = k mod n; Final output s', s.t. s'[l] = s[(i-r)mod n]**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**
    - **Good**: **let n = len(s), r = k mod n; Final output s', s.t. s'[l] = s[(i-r)mod n]**
  - They must be **observable**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**
    - **Good**: **let n = len(s), r = k mod n; Final output s', s.t. s'[I] = s[(i-r)mod n]**
  - They must be **observable**
    - **Bad: Returns None and updates an internal var so the system reads s as rotated**

Specifications

- States what the code **must** do (and not how).
  - They must be **precise**
    - **Bad**: `rotate(s,k)` **handle k naturally and large k are fine**
    - **Good**: **let n = len(s), r = k mod n; Final output s', s.t. s'[I] = s[(i-r)mod n]**
  - They must be **observable**
    - **Bad: Returns None and updates an internal var so the system reads s as rotated**
    - **Good: len(s') = len(s); rotated right by k positions modulo len(s); No mutation of s; raise exception if s is not str**