THE UNIVERSITY OF WESTERN AUSTRALIA

Lecture 19
Exceptions

Revision – Simple Decisions

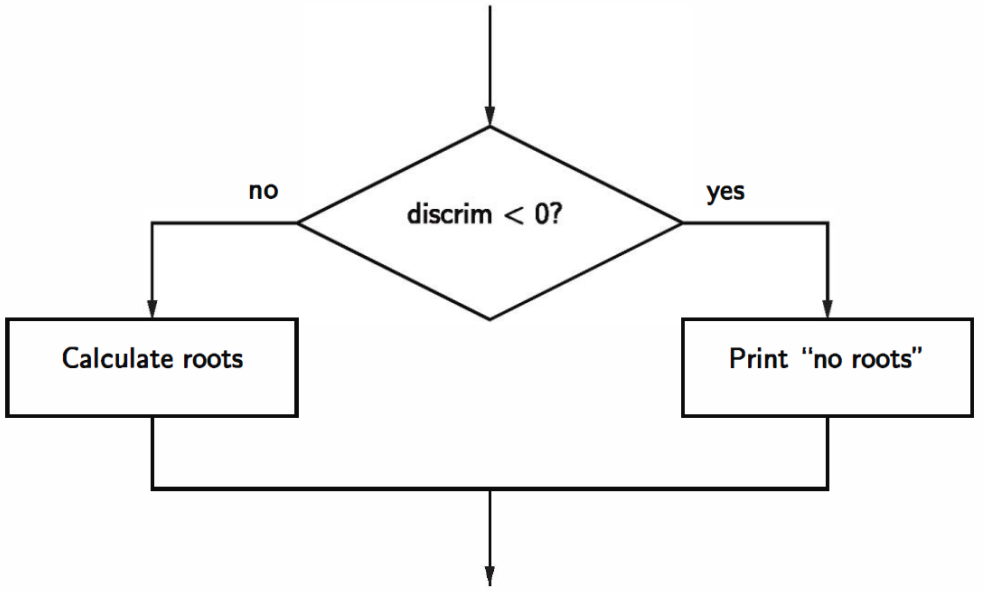
- While sequential flow is a fundamental programming concept, it is not sufficient to solve every problem.
- We need to be able to alter the sequence of a program to suit a particular situation.
- The `if` statement facilitates conditional execution of a code block (if condition evaluates to `True`)

Revision – Two-Way Decisions

- In Python, a two-way decision can be implemented by attaching an else clause onto an if clause.

- This is called an `if-else` statement:

```
if <condition>:
     <statements>
else:
     <statements>
```



Revision – Multi-Way Decisions

- Multi-way decisions are implemented by the if-elif-else construct.
- Allows multiple alternative conditions to be tried, one after the other, until one succeeds – or else if none do.

```
if <condition1>:
   <case1 statements>
elif <condition2>:
   <case2 statements>
elif <condition3>:
   <case3 statements>
…
else:
   <default statements>
```

Exception Handling

- In the quadratic program we used decision structures to avoid taking the square root of a negative number, thus avoiding a run-time error.
- This is true for many programs: decision structures are used to protect against rare but possible errors.
- In the quadratic example, we checked the data *before* making decision. Sometimes functions will check for errors and return a special value to indicate the operation was unsuccessful.
- E.g., a different square root operation might return `-1` to indicate an error (since square roots are never negative, we know this value will be unique).

Exception Handling

```
discRt = otherSqrt(b*b - 4*a*c)
if discRt < 0:
    print("No real roots.“)
else:
    ...
```

- Sometimes programs get so many checks for special cases that the algorithm becomes hard to follow.
  - *Common in code that deals with user supplied inputs*
- Programs that detect an error condition raise an exception
- Programming language designers have come up with a mechanism called exception handling to solve this design problem.

Exception Handling

```
>>> x = int(input("Enter an integer: "))
Enter an integer: a
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with
base 10: 'a'
```

- `int()` is expecting digits so raised a `ValueError` exception when a letter was entered.

Exception Handling

```
>>> print(spam)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'spam' is not defined
```

- In this case spam (without quotes) is treated as a variable, but the variable is not defined
- There is a range of named Exceptions
- https://docs.python.org/3/library/exceptions.html

Exception Handling

- The programmer can write code that catches and deals with exceptions that arise while the program is running, i.e., "Do these steps, and if any problem crops up, handle it this way."
- Often, all that can be done is for the program to exit **gracefully** with a more useful error message

Exception Handling

- The `try` statement has the following form:
-

```
try:
 <body>
except <ErrorType>:
 <handler>
```

- That is, you place the statements that could cause problems within a `try` block
- When Python encounters a `try` statement, it attempts to execute the statements inside the `try` body.
- If exception is raised, execute handler
- If there is no error, control passes to the next statement after the `try .. except`.

Exception Handling

```
# A simple program illustrating chaotic behaviour

def main():
    print("This program illustrates a chaotic function")
    try:
        x = float(input("Enter a number between 0 and 1: "))
    except ValueError:
        print("Non number entered. Cannot proceed")
        return
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(i, x)

main()
```

Exception Handling

- Can have multiple `except` blocks. Multiple `except` statements act like `elif`. If an error occurs, Python will test each except looking for one that matches the type of error.
- A bare `except` acts like an `else` and catches any errors without a specific exception type.
- If there was no bare `except` at the end and none of the `except` clauses match, the program would still crash and report an error.

Exception Handling Style

- Exceptions are intended for *exceptional* circumstances
- Exceptions should not be used as a substitute for carefully planned `if` statements

Exception Handling

```
def main():
    print("This program illustrates a chaotic function")
    try:
        x = float(input("Enter a number between 0 and 1: "))
    except ValueError:
        print("Non number entered. Cannot proceed")
        return
    if x > 0 and x < 1.0 :
        for i in range(10):
            x = 3.9 * x * (1 - x)
            print(i, x)
    else:
        print("number entered was not in range 0 .. 1.0”)
main()
```

Exception Handling

- A very common use for exception handling is when opening files, which may fail for a range of reasons, many of which are hard to predict and tedious to test for.
  - *File not present or empty (reading)*
  - *Directory permissions do not allow file creation for writing*
  - *File permissions do not allow reading or appending*

Exception Handling – File Opening

```
# Opens named file with specified mode, returning the file handle
# If that results in an exception, message printed, and None returned
def test_fileIO(name, mode) :
    mode_list = [“r”, “w”, “a”]
    if mode not in mode_list:
        print("Unknown file access mode", mode)
        return(None)
    try:
        f = open(name, mode)
    except IOError:
        print(f“cannot open the file {name:s}”)
        return(None)
    return(f)
```

Summary

- In this lecture we learnt about:
  - *Exceptions as unexpected situations and how to handle them in Python*
  - *Increment assignment*