Chapter 17 Data Validation and Verification

Contents

Syllabus Learning Outcomes

2.4 Data Validation and Program Testing
Use data validation techniques and design test cases
- 2.4.1 Explain the difference between data validation and data verification.
- 2.4.2 Understand data validation techniques such as:
  - – range check
  - – format check
  - – length check
  - – presence check
  - – check digit.
- 2.4.3 Identify, explain and correct syntax, logic, and runtime errors.
- 2.4.4 Design appropriate test cases using normal, abnormal and extreme data for testing and debugging programs.

1 Data validation and verification

Data integrity, which is the accuracy or validity of data, can never be guaranteed. However, the chances are improved if appropriate measures are taken when data originally enters a system or when it is transmitted from one system to another.

1.1 Types of input error

Users keying in data could make **transcription errors,** a single incorrect character being entered, such as a wrong password or an invalid email address. For eg, a correct postal code *449035* could be wrongly entered as *449036* or the name *Stanley* might be wrongly entered as *Stamley* instead.

Another type of input error is **transposition error**, where two digits are accidentally swapped. For eg, a correct NRIC *S8312345B* is wrongly entered as *S8312354B*.

1.2 Validation of data entry

Data validation is a process of ensuring that the input data supplied to a system satisfies a set of rules such that it is sensible, complete, and within acceptable boundaries. Its purpose is to avoid data errors. It does not guarantee that data is accurate. For example, if the entry of a name is expected but the wrong name is entered, it will still be recognised as a name and therefore accepted as valid.

Data validation techniques include:

- **range check** – Checks that a value falls within the specified range, e.g. The marks of the A level H2 Computing Paper 1 must fall within the specified range of 0 to 100 and not anything less than 0 or more than 100 marks.

- **format check** – Checks the data is in the right format, e.g. A date has to be dd/mm/yyyy.

- **length check** – Checks the data isn't too short or too long, e.g. A telephone number must have a length of 8 numbers.

- **presence check** – Checks that data has been entered into a field, e.g. ensure that an entry field is not left blank.

- **check digit** – An extra digit added to the end of a code number, which has been calculated from the digits of the code number. This ensures the code number has been entered properly.

1.2.1 Check digit

Code numbers, which could be made up of either digits only or a combination of digits and letters) such as customer number, employee number, or product number are often lengthy and prone to error when being keyed in. One way of detecting these errors is to add a check digit to the end of the code numbers.

A check digit is a digit (or character) added to the end of a code number which has been calculated from the digits (or characters) of the code number. In this way the code number with its extra check digit is self-checking.

Some examples of real-life data that use check digits are

- Singapore NRIC
- Singapore car plate registration number
- book ISBN
- credit card number (e.g. Luhn algorithm)

1.2.2 Method of calculating check digit

The best-known method of calculating check digit is the modulo 11 system, which traps over 99% of all errors. The calculation of a check digit is shown below:

1. Each digit of the code number is assigned a 'weight'. The right-hand (least significant) digit is given a weight of 2, the next digit to the left 3, and so on.

2. Each digit is multiplied by its weight and the products are added together.

3. The sum of the products is divided by 11 and the remainder is obtained.

4. The remainder is subtracted from 11.

5. The result is divided again by 11 and the remainder (results mod 11) is the check digit. The only exception is that if the remainder is 10 and only a single check digit is allowed, the Roman number 'X' is used as a check digit.

Example: **ISBN check digit**

Digits: 3 9 2 8 4 4 4 0 4

Weight: 10 9 8 7 6 5 4 3 2

Results: 30 + 81 + 16 + 56 + 24 + 20 + 16 + 0 + 8 = 251

Calculate remainder: 251 / 11 = 22 Remainder 9 -> 11 - 9 = 2

Check digit 2 **mod** 11 = 2

The complete ISBN code number is 3928444042.

Steps to verify if the check digit included is correct:

- The entire number is multiplied by the same weights that were used to calculate and the check digit itself is multiplied by 1.
- The results of the multiplication are added together.
- The number is correct if the sum is divisible by 11 with a remainder 0.

In conclusion, the check digit should be a number that is appended to the string of code number, to make a sum of products that, when divided by 11, gives no remainder.

Using the above ISBN code number to check:

Digits: 3 9 2 8 4 4 4 0 4 2

Weight: 10 9 8 7 6 5 4 3 2 1

Results: 30 + 81 + 16 + 56 + 24 + 20 + 16 + 0 + 8 + 2 = 253

Calculate remainder: 253 / 11 = 23 Remainder 0

1.3 Verification of data entry

Verification is the process of getting the user to confirm that the data entered was what was intended to be entered. Essentially, it is a way of preventing errors when data is copied from one medium to another. It does not check if the data makes sense or is within acceptable boundaries. It only checks that the data entered is identical to the original source.

There are two methods of verification.

Double entry is one method of verification. The process of entering data twice, with the second entry being compared with the first is to ensure that it is accurate. It is common in batch processing for a second data entry operator to key in a batch of data to verify it. Another example is when setting a password, a user would be asked to key in the password a second time to ensure that the passwords match up.

Proofreading data is another method that involves someone checking the data entered against the original source to ensure that the data match.

1.4 Workflow for data validation

As the data is being keyed in, a computer program controlling the input can perform various validation checks on the data by

- having a prompt to get user input
- apply appropriate data validation techniques on user input
- output relevant error message for invalid data and allow re-entry until user input valid data

Example of data validation:

User input: Age
Valid range for age: 1 – 100

Pseudocode:

```
valid_age <- false
REPEAT
    INPUT age
    IF empty input
        OUTPUT empty input message and allow re-entry
    ELIF not a number
        OUTPUT wrong data type and allow re-entry
    ELIF not within valid range
        OUTPUT invalid range and allow re-entry
    ELSE
        set valid_age to true
    END IF
UNTIL valid_age is true
```

2 Program testing

Before a piece of software can be used, it needs to be **tested**. Some testing is done by the programmer as the code is written. Other testing is more structured and involves a **test plan** being written and the software being tested with a variety of types of data to check that the outputs are as expected. Errors or mistakes in a program are often referred to as bugs. The process of finding and eliminating errors is called **debugging**.

Errors can be classified into three major groups:

- Syntax errors
- Logical errors
- Runtime errors

2.1 Syntax errors

Syntax errors are errors that occur when one has not followed the rules of the language. A program with syntax errors will not run.

Common causes of Python syntax errors:

- Spelling or typing errors
- Missing parentheses, ( ) or { }
- Missing colons (:) or semicolons (;) in statements in which they are required by the language
- Missing or unexpected indentation in Python

- Printing a value without declaring it

2.2 Logical errors

Logical errors are the most difficult to fix. A program with logic errors will execute, but it will not behave as expected, producing an incorrect output. The error is caused by a mistake in the program's logic. You won't get an error message because no syntax or runtime error has occurred. You will have to find the problem by reviewing all the relevant parts of the code.

Sometimes there can be absolutely nothing wrong with your Python implementation of an algorithm – the algorithm itself can be incorrect. However, more frequently these kinds of errors are caused by programmer carelessness. Here are some examples of mistakes that lead to logical errors:

Common causes of Python logic errors:

- using the wrong variable name
- indenting a block to the wrong level
- Not correctly understanding what the program needed to do
- Using the incorrect logical operator in a selection statement
- Missing or incorrect positioning of brackets in mathematical calculations, which means that the incorrect result is returned
- Loops that execute more or fewer times than intended

2.3 Runtime errors

Runtime errors are errors which cause a program to crash while it is executing. These errors occur when the code is valid according to rules of syntax and semantics (ensuring that a compiler can understand the source code), and is also logically correct, but the environment that the program is being run in or the input provided causes an error to occur.

Common causes of Python runtime errors:

- A mathematical calculation that results in the program trying to divide by 0
- Performing an operation on incompatible types
- Using an identifier that has not been defined
- Accessing a list element, dictionary value, or object attribute which doesn't exist
- Linking to a file or resource that has been moved or no longer exists

2.3.1 Exception Handling

Run-time errors can occur for many reasons as stated above. These errors are called 'exceptions'. Python distinguishes between different types of exceptions, such as and not limited to

- IOError: for example, a file cannot be opened
- ImportError: Python cannot find the module
- ValueError: an argument has an inappropriate value
- EOFError: a file-read meets an end-of-file condition
- ZeroDivisionError: a division by zero has been attempted

The exceptions can be resolved with an exception handler, rather than let the program crash.

```
Psuedocode:

TRY
    <statementsA>
EXCEPT
    <statementsB>
ENDTRY
```

Catching specified exceptions

```python
try:
    result = 100/2
    print(result)
except ZeroDivisionError:
    print("No division by zero!")
```

```
50.0
```

```python
# Multiple exceptions

x = 10
try:
    result = 100/0
    print(y)
except ZeroDivisionError:
    print("No division by zero!")
except ValueError:
    print("There is an error in the value")
except TypeError:
    print("There is a type error")

```

```
No division by zero!
```

```
x = 10
try:

    result = int('abc')

    print(y)
except ZeroDivisionError:
    print("No division by zero!")
except (ValueError, TypeError):
    print("An error occurred")

```

```
An error occurred
```

Catching exceptions

```
randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except Exception as e:
        print("Oops!", repr(e), "occurred.") # The repr() function returns a printable representation of the given object.
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

```
The entry is a
Oops! ValueError("invalid literal for int() with base 10: 'a'",) occurred.
Next entry.

The entry is 0
Oops! ZeroDivisionError('division by zero',) occurred.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

Try, except, finally

```
#The try block will raise an error when trying to write to a read-only file:

try:
    f = open("demofile.txt")
    f.write("Lorum Ipsum")
except Exception as e:
    print("Oops!", repr(e), "occurred.")
    print("Something went wrong when writing to the file.")
    print()
finally:
    print("Closing file.")
    f.close()

#The program can continue, without leaving the file object open
```

```
Oops! UnsupportedOperation('not writable',) occurred.
Something went wrong when writing to the file.

Closing file.
```

Raising an Exception

We can use raise to throw an exception if a condition occurs. The program comes to a halt and displays the exception, offering clues about what went wrong.

```
x = 'a'

if type(x) != int:
    raise TypeError("Only integers are allowed!")

y = x + 2
print(y)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-f77d69be6dda> in <module>()
      2
      3 if type(x) != int:
----> 4     raise TypeError("Only integers are allowed!")
      5
      6 y = x + 2

TypeError: Only integers are allowed!
```

2.4 Test data

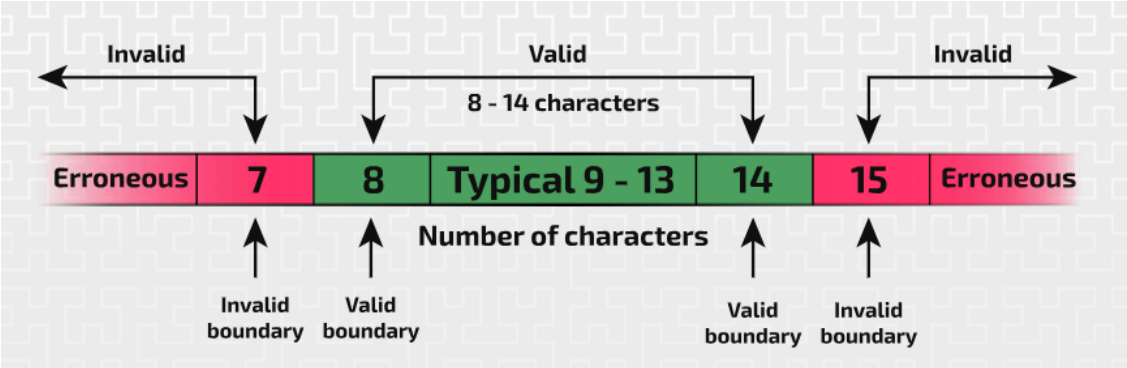
There are 3 main types of test data.

Type of data	Description
Normal/valid	Typical data values that are valid and will be accepted by the system
Abnormal/invalid/erroneous	Data values that the system should not accept or should be rejected
Extreme/boundary	Data values that are at the extreme end of the range of normal data that should be accepted

Example: Checking the length of a password

The description of the length of a password is as follows:

- Passwords that have fewer than 8 or more than 14 characters would be considered invalid
- Passwords that have 7 or 15 characters would be considered invalid as these lengths are just outside the accepted limits
- Passwords that have 8 or 14 characters would be considered valid and boundary
- Finally, passwords that have between 9 and 13 characters would be considered valid and normal or typical



It is important to test using a variety of test data to ensure that the system is robust and works as expected.

Annex 1: Datetime

One often needs to manage dates in programs, eg date of creation/update, loan/due date

Common tasks include:

- generate the current system date
- read date from keyboard/file
- date arithmetic
- write date to display/file

```
# datetime module has many useful functions to manipulate date and time, so use it.
import datetime
```

Get current date and time using datetime class

```
# current system date and time
current_datetime = datetime.datetime.now()

print(current_datetime)
print(type(current_datetime))
```

```
2023-11-07 22:35:36.120643
<class 'datetime.datetime'>
```

Print year, month, day

```
print(current_datetime.year)
print(current_datetime.month)
print(current_datetime.day)
```

```
2023
11
7
```

Print time

```
print(current_datetime.hour)
print(current_datetime.minute)
print(current_datetime.second)
print(current_datetime.microsecond)
```

```
22
35
36
120643
```

Get today's date using date class

```
# current system date
today = datetime.date.today()

print(today)
print(type(today))
# current date is a datetime date object giving year, month and day eg datetime.date(2021, 3, 15)
```

```
2023-11-07
<class 'datetime.date'>
```

Print today's year, month, day

```
print(today.day)
print(type(today.day))
print(today.month)
print(today.year)
```

```
7
<class 'int'>
11
2023
```

Initiate a date object

```
d = datetime.date(2050, 4, 13)
print(d)
```

```
2050-04-13
```

Formatting dates

```
# return date as string in the format YYYY-MM-DD
print(today.isoformat())

print('2020-03-26' > today.isoformat())
```

```
2023-11-07
False
```

```
# can find out more at https://www.w3schools.com/python/python_datetime.asp
# conversion to different formats
# %d - dd
# %m - mm
# %y - yy
# %Y - YYYY
# %D - dd/mm/yy
# %A - day spelt in full
# %a - day spelt in short form
# %B - month spelt out in full
# %b - month spelt in short form
print(today.strftime('%d-%m-%y'))
print(today.strftime('%d-%m-%Y'))
print(today.strftime('%d%m%Y'))
print(today.strftime('%Y%m%d'))
print(today.strftime('%D'))
print(today.strftime('%a'))
print(today.strftime('%b'))
```

```
28-03-21
28-03-2021
28032021
20210328
03/28/21
Sun
Mar
```

```
# days of week and month
print(today.strftime('%A %d %B %Y'))
print(today.strftime('%a %d %b %Y'))
```

```
Sunday 28 March 2021
Sun 28 Mar 2021
```

Exception handling for date object

```python
# check for valid date format
date = input("Enter date in DD-MM-YYYY: ")

#strptime() method creates a datetime object from the given string
try: # normal processing
    valid_date = datetime.datetime.strptime(date, "%d-%m-%Y") # p - parse
    print(valid_date)
    print("date ok")
except ValueError:
    # if error happens
    print("Invalid date format")

```

```
Enter date in DD-MM-YYYY: 05-13-2022
Invalid date format
```

Annex 2: Timedelta

Arithmetic operations on datetime object gives timedelta object

```python
import datetime

day1 = datetime.date.today()
day2 = datetime.date(2023, 5, 20)
print(type(day1))

print("Day 1:",day1)
print("Day 2:", day2)

difference = day1 - day2
print("Difference between day 1 and day 2:",difference)

# difference between datetime object gives timedelta object
print(type(difference))

# timedelta.days gives number of days
print(difference.days)
```

```
<class 'datetime.date'>
Day 1: 2023-11-11
Day 2: 2023-05-20
Difference between day 1 and day 2: 175 days, 0:00:00
<class 'datetime.timedelta'>
175
```

Calculation of new date

```python
# method to calculate new date
today = datetime.date.today()
print("Today:", today)

#calculating new date for 2 days after today
new_date = day1 + datetime.timedelta(days = 2)
print("New date:",new_date)
```

```
Today: 2023-11-11
New date: 2023-11-13
```

Generation of random date

```
# generate random date

import datetime, random

start_date = datetime.date(2023, 1, 1)
end_date = datetime.date(2023, 12, 31)

time_difference = end_date - start_date
print("Time difference: ",time_difference)
print(type(time_between_dates))

days_between_dates = time_between_dates.days #get number of days between start and end date
print("Day difference:",days_between_dates)
print(type(days_between_dates))

random_number_of_days = random.randrange(days_between_dates)
random_date = start_date + datetime.timedelta(days=random_number_of_days)
print("Random date:",random_date)
```

```
Time difference:  364 days, 0:00:00
<class 'datetime.timedelta'>
Day difference: 364
<class 'int'>
Random date: 2023-05-31
```

Annex 3: Regular expression

- A powerful mechanism to check against patterns
- Used in data validation to avoid complicated (nested) if-else statements
- Refer to https://docs.python.org/3.6/library/re.html for more information

Example 1: gender validation

```
#https://docs.python.org/3.6/library/re.html

# gender validation
import re

# get gender
gender = input("Enter gender: ")

# accept both upper and lowercase f and m
pattern = re.compile("[fFmM]")

if pattern.match(gender):
    print("Cool! You are not in between")
else:
    print("Oh. What are you then?")
```

```
Enter gender: p
Oh. What are you then?
```

Example 2: NRIC format validation

```
# # NRIC validation
import re

# get nric
nric = input("Enter NRIC: ")

# ^ - starts with, {n} - exactly n times, $ - ends with
pattern = re.compile("^[sStTfFgG][0-9]{7}[a-zA-Z]$")

if pattern.match(nric):
  print("Welcome to Singapore!")
else:
  print("You have entered an invalid NRIC!")
```

```
Enter NRIC: W1234)
You have entered an invalid NRIC!
```