COMP1531

2.1 - Requirements

Updates

Lecture Code

Lecture code available at:

https://gitlab.cse.unsw.edu.au/COMP1531/19T3-lectures

Lab reminder

Don't forget that lab submissions are in two parts:

1. Submission of the lab via "1531 submit" on Sunday 5pm the week it was released
2. Getting it checked off in-person (manually) with your tutor in the lab the week it was released, or the week after

You must complete both of these to be awarded the marks

Python knowledge

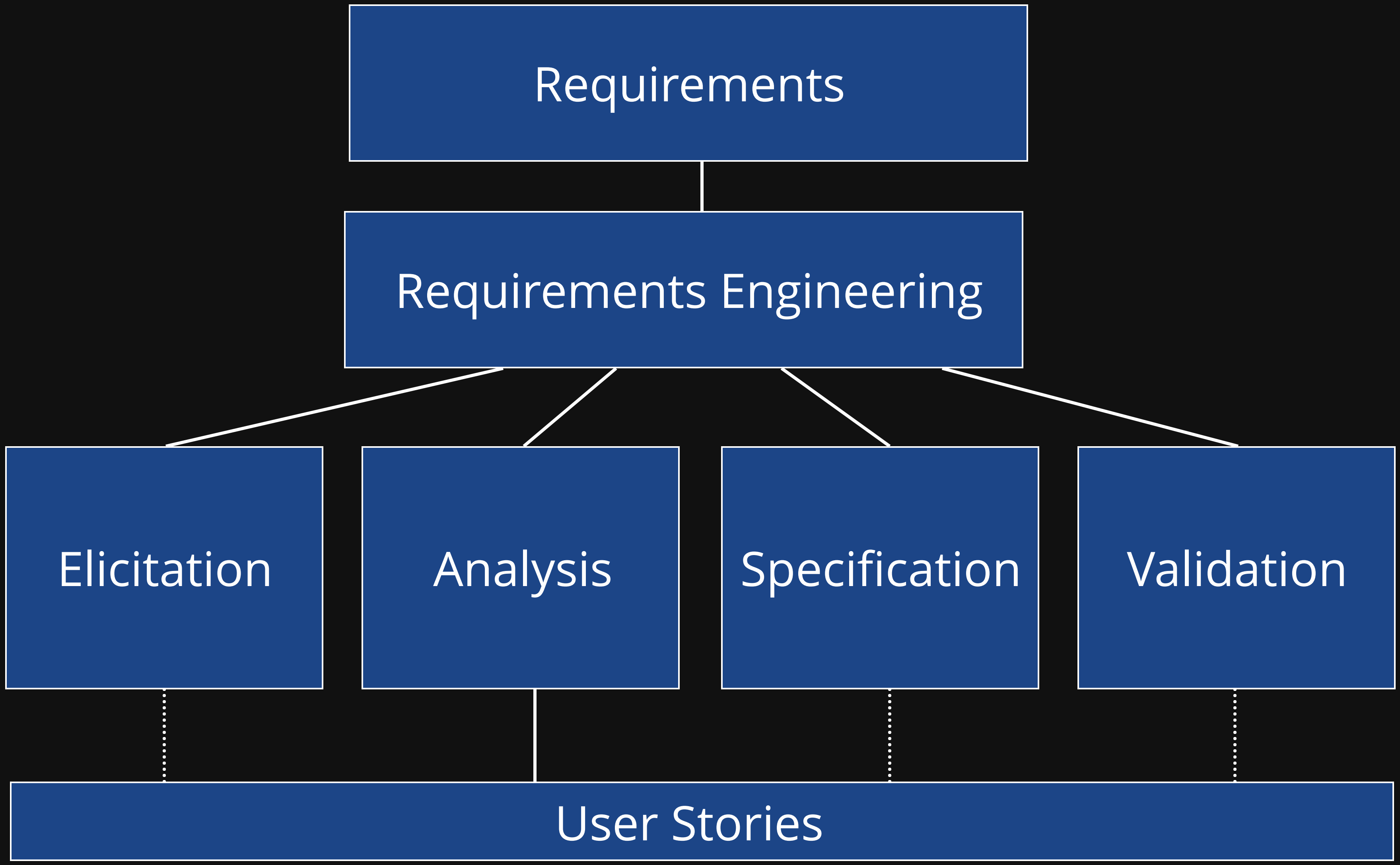
We will teach you enough python to complete the activities.

But this is barely the surface. We strongly encourage you to do your own reading into Python.

SDLC


REQUIREMENTS
ANALYSIS
DESIGN
MAINTENANCE
SDLC
PHASES
DEVELOPMENT
DEPLOYMENT
TESTING

Requirements



Requirements

IEEE defines a requirement as:

A condition or capability needed by a user to solve a problem or achieve an objective

We would also describe requirements as:

- Agreement of work to be completed by all stakeholders
- Descriptions and constraints of a proposed system

Functional v Non-Functional

Functional requirements specify a specific capability/service that the system should provide.

Non-functional requirements place a constraint on *how* the system can achieve that. Typically this is a performance characteristic.

Functional v Non-Functional

For example:

Functional: The system must send a notification to all users whenever there is a new post, or someone comments on an existing post

Non-functional: The system must send emails no later than 30 minutes after from such an activity

Requirements Engineering

We need a durable process to determine requirements

“The hardest single part of building a software system is deciding what to build. No part of the work so cripples the resulting systems if done wrong” (Brooks, 1987)

Requirements Engineering

Requirements Engineering is:

- A **set of activities** focused on identifying the purpose and goal of a software system
- A **negotiation process** where stakeholders agree on what they want. Stakeholders include:
  - End user(s)
  - Client(s) (often businesses)
  - Design team(s)

Requirements Engineering

Requirements engineering often follows a logical process across 4 steps:

1. Elicitation of raw requirements from stakeholders
2. Analysis of requirements
3. Formal specification of requirements
4. Validation of requirements

RE | Step 1 | Elicitation

Questions and discovery

- Market Research
- Interviews with Stakeholders
- Focus groups
- Asking questions "What if? What is?"

RE | Step 2 | Analysis

Building the picture

- Identify dependencies, conflicts, risks
- Establish relative priorities
- Usually done through:
  - User stories (discussed today)
  - Use cases (discussed next week)

RE | Step 3 | Specification

Refining the picture

- Establishing the right sense of granularity
  - There is no perfect way to granulate
- Often the stage of breaking up into functional and non-functional
- E.G. Try and granulate "The system shall keep the door locked at all times, unless instructed otherwise by an authorised user. When the lock is disarmed, a countdown shall be initiated at the end of which the lock shall be automatically armed (if still disarmed)"

RE | Step 4 | Validation

Going back to stakeholders and ensuring requirements are correct

Challenges during RE?

What are some challenges we may face while engaging in Requirements engineering?

Challenges during RE?

What are some challenges we may face while engaging in Requirements engineering?

- Requirements sometimes only understood after design/build has begun
- Clients/customers sometimes don't know what they want
- Clients/customers sometimes change their mind
- Developers might not understand the subject domain
- Limited access to stake holders
- Jumping into details or solutions too early (XY problem)

What matters?

- Investigate stakeholder needs
- Expand, refine, and connect *specific* ideas
- Understand the iterative and ongoing nature
    - Humans are imperfect

User Stories - Overview

User Stories are a method of requirements engineering used to inform the development process and what features to build.

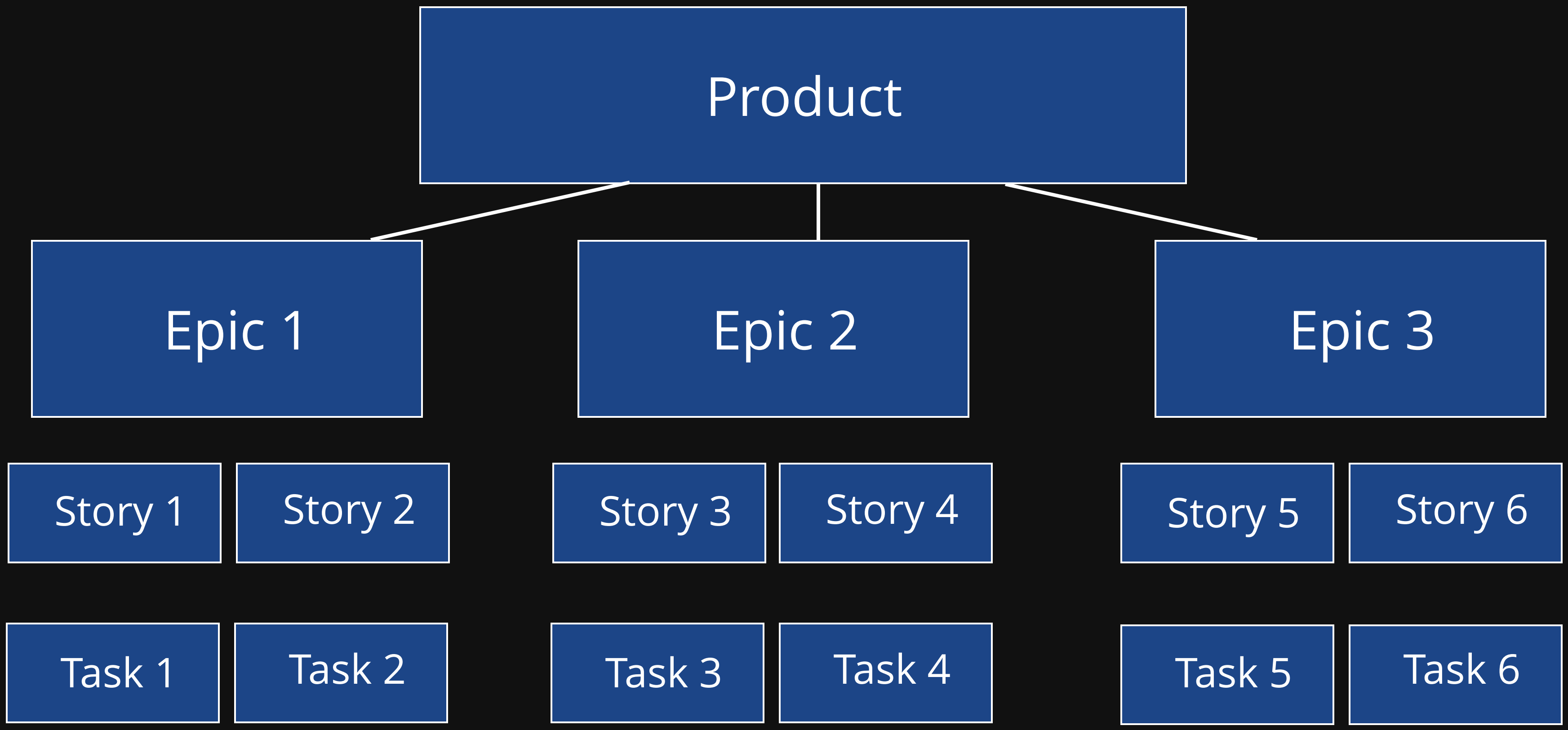
User Stories - Structure

When a customer tells you what they want, try and express it in the form **As a < type of user >, I want < some goal > so that < some reason >**

E.G. They say:

- E.G. They say:
  - A student can purchase monthly parking passes online
- But your story becomes:
  - As a **student**, I want to **purchase a parking pass** so that **I can drive to school**

User Stories - Structure



User Stories - Nature

User stories:

- Are written in non-technical language
- Are user-goal focused, not product-feature focused
  - User stories inform feature decisions

Why do we care?

- The keep customers at the centre
- Keep it problem focused, not solution focused

User Stories - Activity

Let's design a bag.

Or a to-list.

Or anything.

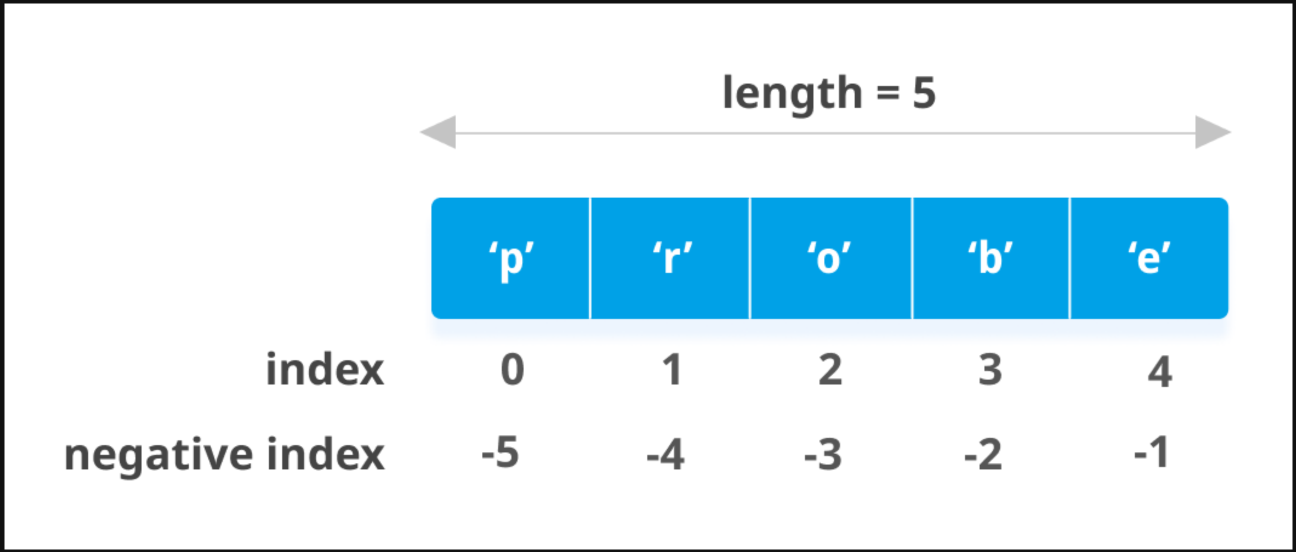
User Stories - More

Read more about user stories here:

https://www.atlassian.com/agile/project-management/user-stories

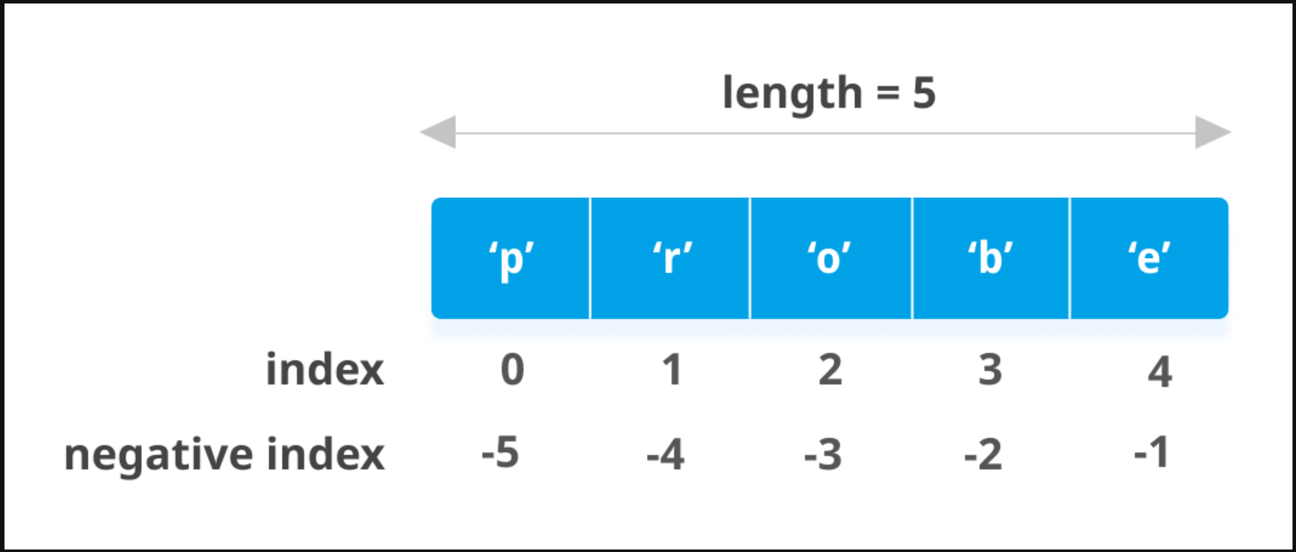
Python - Dictionaries

Lists are **sequential containers** of memory. Values are referenced by their **integer index** (key) that represents their location in an **order**



Python - Dictionaries

Lists are **sequential containers** of memory. Values are referenced by their **integer index** (key) that represents their location in an **order**



Python - Dictionaries

Dictionaries are **associative containers** of memory. Values are referenced by their **string key** that *maps* to a value

name → "sally"

age → 18

height → "187cm"

Python - Dictionaries

Dictionaries are **associative containers** of memory. Values are referenced by their **string key** that *maps* to a value

dict_basic_1.py

```
userData = {}
userData["name"] = "Sally"
userData["age"] = 18
userData["height"] = "187cm"
print(userData)
```

```
{'name': 'Sally', 'age': 18, 'height': '187cm'}
```

Python - Dictionaries

There are a number of different ways we can construct and interact with dictionaries

dict_basic_2.py

```
userData = {
   'name' : 'Sally',
   'age' : 18,
   'height' : '186cm', # Why a comma?
}
userData['height'] = '187cm'
print(userData)
```

```
{'name': 'Sally', 'age': 18, 'height': '187cm'}
```

Python - Dictionaries

dict_loop.py

Basic loops are over **keys** not **values:**

How would we modify this to print out the values instead?

```python
userData = [
    {
        'name' : 'Sally',
        'age' : 18,
        'height' : '186cm',
    }, {
        'name' : 'Bob',
        'age' : 17,
        'height' : '188cm',
    },
]
for user in userData:
    print("Whole user: ", user)
    for part in user:
        print(f"  {part}")
```

```
Whole user:  {'name': 'Sally', 'age': 18, 'height': '186cm'}
  name
  age
  height
Whole user:  {'name': 'Bob', 'age': 17, 'height': '188cm'}
  name
  age
  height
```

Python - Dictionaries

dict_loop_2.py

```python
userData = {'name' : 'Sally','age' : 18, \
            'height' : '186cm'}

for user in userData.items():
    print(user)
print("====================")

for user in userData.keys():
    print(user)

print("====================")
for user in userData.values():
    print(user)
```

```
('name', 'Sally')
('age', 18)
('height', '186cm')
====================
name
age
height
====================
Sally
18
186cm
```

Python - Dictionaries

Q. Write a python program that takes in a series of words from STDIN and outputs the frequency of how often each vowel appears

Python - Exceptions

An **exception** is an action that disrupts the normal flow of a program. This action is often representative of an error being thrown. Exceptions are ways that we can elegantly recover from errors

Python - Exceptions

The simplest way to deal with problems...

Just crash

exception_1.py

```python
import sys

def sqrt(x):
    if x < 0:
        sys.stderr.write("Error Input < 0\n")
        sys.exit(1)
    return x**0.5

if __name__ == '__main__':
    print("Please enter a number: ",)
    inputNum = int(sys.stdin.readline())
    print(sqrt(inputNum))
```

Python - Exceptions

Now instead, let's raise an exception

However, this just gives us more information, and doesn't help us handle it

exception_2.py

```
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Error, sqrt input {x} < 0")
    return x**0.5

if __name__ == '__main__':
    print("Please enter a number: ",)
    inputNum = int(sys.stdin.readline())
    print(sqrt(inputNum))
```

Python - Exceptions

Now instead, let's raise an exception

However, this just gives us more information, and doesn't help us handle it

exception_2.py

```
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Error, sqrt input {x} < 0")
    return x**0.5

if __name__ == '__main__':
    print("Please enter a number: ",)
    inputNum = int(sys.stdin.readline())
    print(sqrt(inputNum))
```

Python - Exceptions

If we catch the exception, we can better handle it

exception_3.py

```
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Error, sqrt input {x} < 0")
    return x**0.5

if __name__ == '__main__':
    try:
        print("Please enter a number: ",)
        inputNum = int(sys.stdin.readline())
        print(sqrt(inputNum))
    except Exception as e:
        print(f"Error when inputting! {e}. Please try again:")
        inputNum = int(sys.stdin.readline())
        print(sqrt(inputNum))
```

Python - Exceptions

Or we could make this even more robust

exception_4.py

```
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Error, sqrt input {x} < 0")
    return x**0.5

if __name__ == '__main__':
    print("Please enter a number: ",)
    while True:
        try:
            inputNum = int(sys.stdin.readline())
            print(sqrt(inputNum))
            break
        except Exception as e:
            print(f"Error when inputting! {e}. Please try again:")
```

Python - Exceptions

Key points:

- Exceptions carry data
- When exceptions are thrown, normal code execution stops

throw_catch.py

```
import sys

def sqrt(x):
    if x < 0:
        raise Exception(f"Input {x} is less than 0. Cannot sqrt a number < 0")
    return x**0.5

if __name__ == '__main__':
    if len(sys.argv) == 2:
        try:
            print(sqrt(int(sys.argv[1])))
        except Exception as e:
            print(f"Got an error: {e}")
```

Python - Exceptions

Examples with pytest (very important for project)

pytest_except_1.py

```
import pytest

def sqrt(x):
    if x < 0:
        raise Exception(f"Input {x} is less than 0. Cannot sqrt a number < 0")
    return x**0.5

def test_sqrt_ok():
    assert sqrt(1) == 1
    assert sqrt(4) == 2
    assert sqrt(9) == 3
    assert sqrt(16) == 4

def test_sqrt_bad():
    with pytest.raises(Exception, match=r"*Cannot sqrt*"):
        sqrt(-1)
        sqrt(-2)
        sqrt(-3)
        sqrt(-4)
        sqrt(-5)
```

Python - Exception Sub-types

Other basic exceptions can be caught with the "Exception" type

pytest_except_2.py

```
import pytest

def sqrt(x):
    if x < 0:
        raise ValueError(f"Input {x} is less than 0. Cannot sqrt a number < 0")
    return x**0.5

def test_sqrt_ok():
    assert sqrt(1) == 1
    assert sqrt(4) == 2
    assert sqrt(9) == 3
    assert sqrt(16) == 4

def test_sqrt_bad():
    with pytest.raises(Exception):
        sqrt(-1)
        sqrt(-2)
        sqrt(-3)
        sqrt(-4)
        sqrt(-5)
```

Project

Project iteration 1has been released:

- pytest
- User Stories