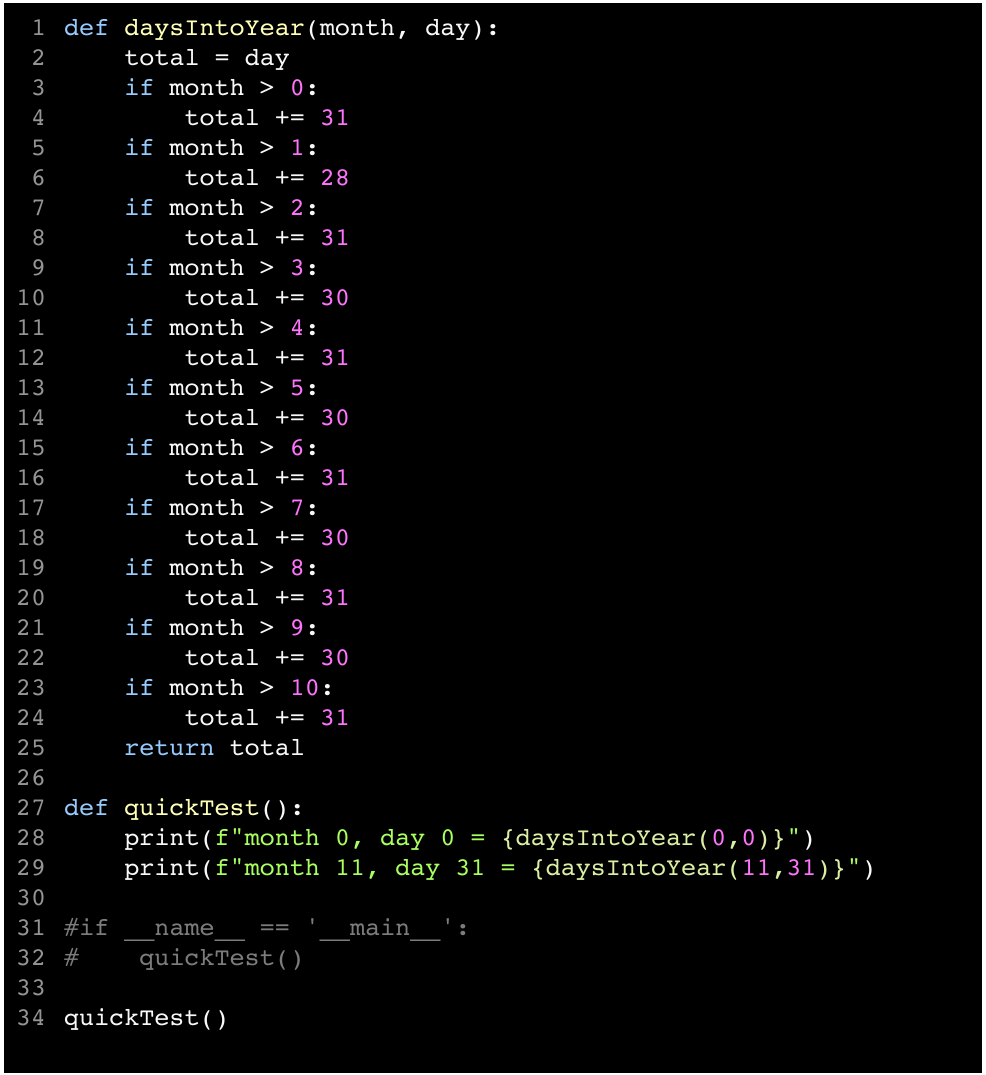
COMP1531

2.2 - Dictionaries & Exceptions

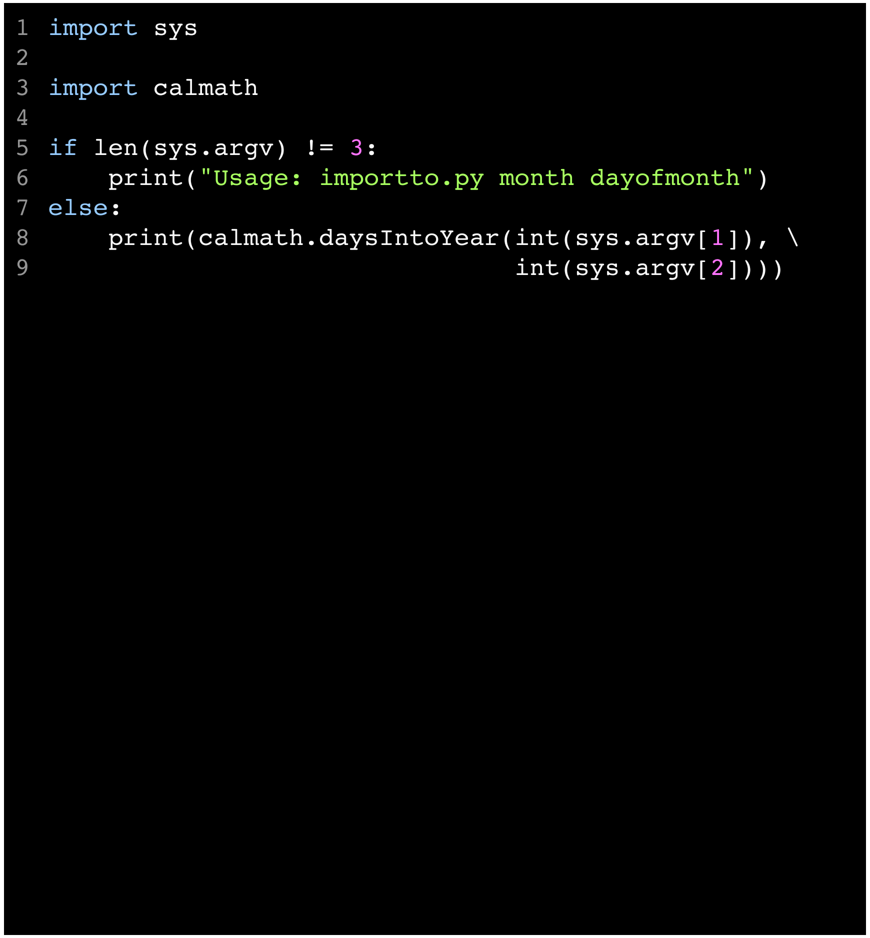
importing and modules

calmath.py



```python
def daysIntoYear(month, day):
    total = day
    if month > 0:
        total += 31
    if month > 1:
        total += 28
    if month > 2:
        total += 31
    if month > 3:
        total += 30
    if month > 4:
        total += 31
    if month > 5:
        total += 30
    if month > 6:
        total += 31
    if month > 7:
        total += 30
    if month > 8:
        total += 31
    if month > 9:
        total += 30
    if month > 10:
        total += 31
    return total

def quickTest():
    print(f"month 0, day 0 = {daysIntoYear(0,0)}")
    print(f"month 11, day 31 = {daysIntoYear(11,31)}")

#if __name__ == '__main__':
#    quickTest()

quickTest()
```

importto.py



```python
import sys

import calmath

if len(sys.argv) != 3:
    print("Usage: importto.py month dayofmonth")
else:
    print(calmath.daysIntoYear(int(sys.argv[1]), \
                               int(sys.argv[2])))
```

"testpath" example

See week 2 lecture code

Python Path

This is something needed to make pytest work

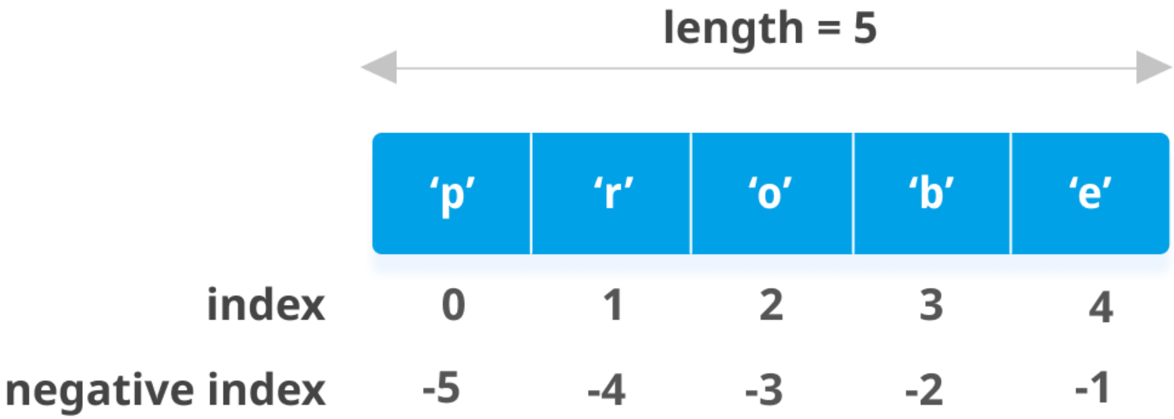
If your project is in ~/cs1531/project

```
export PYTHONPATH="$PYTHONPATH:~/cs1531/project"
```

You can add this line to your ~/.bashrc if you don't want to type it in every time you open a terminal

Python - Dictionaries

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

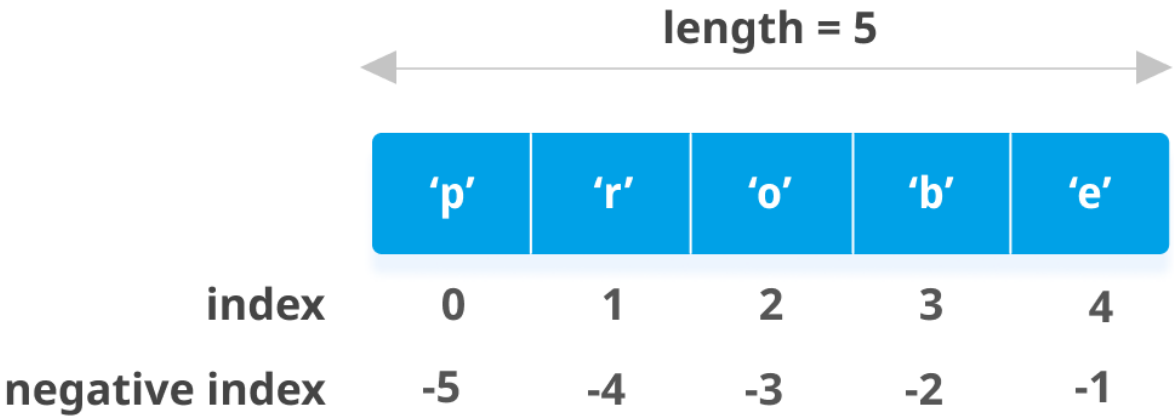


length = 5

	‘p’	‘r’	‘o’	‘b’	‘e’
index	0	1	2	3	4
negative index	-5	-4	-3	-2	-1

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

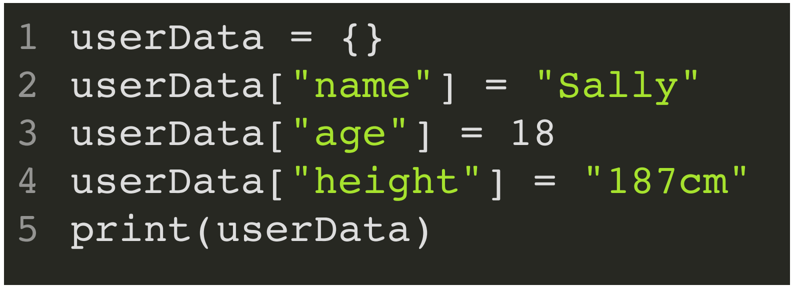
Dictionaries are **associative containers** of memory. Values are referenced by their **string key** that *maps* to a value

name	"sally"
age	18
height	"187cm"

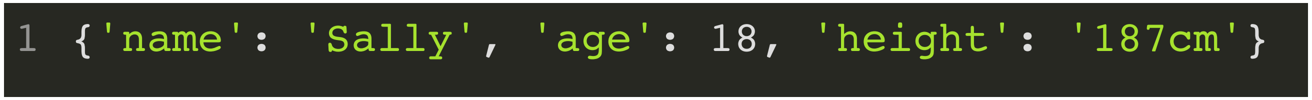
Python - Dictionaries

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

dict_basic_1.py



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

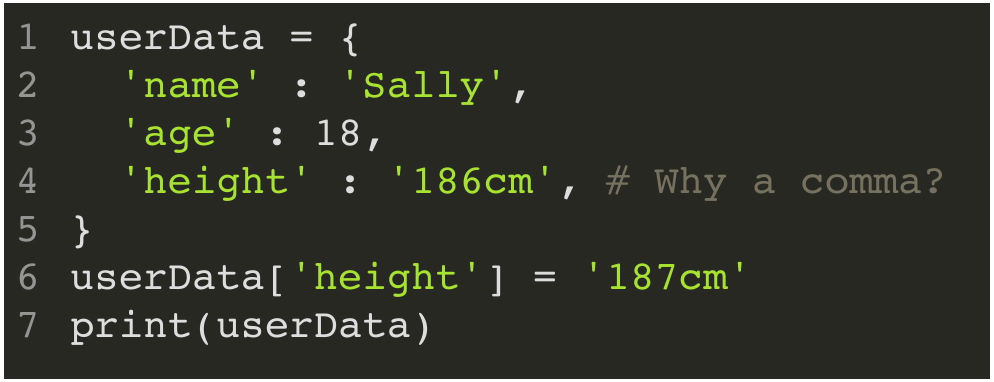


```
{'name': 'Sally', 'age': 18, 'height': '187cm'}
```

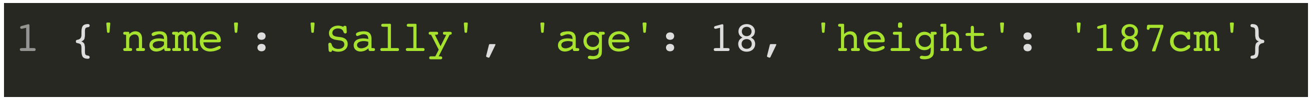
Python - Dictionaries

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

dict_basic_2.py



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



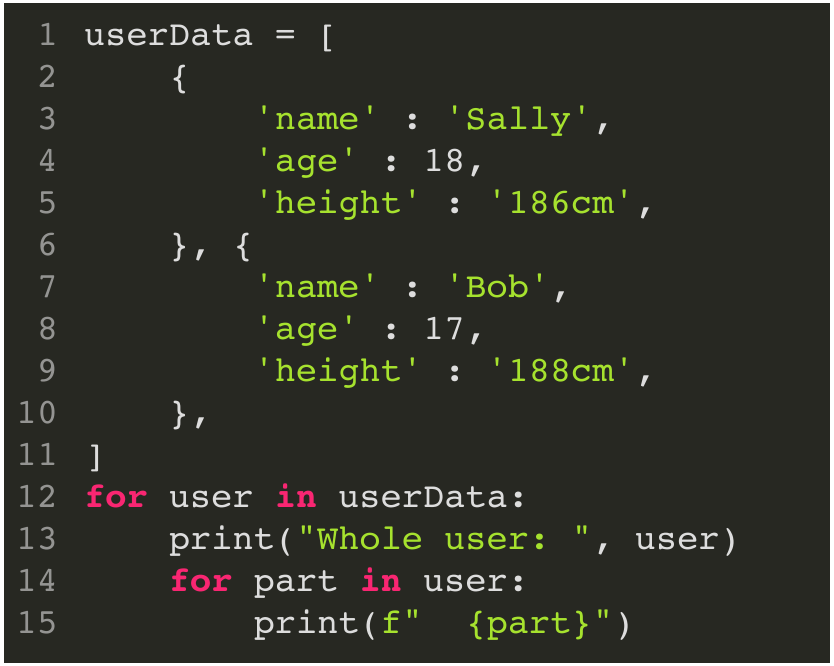
```
{'name': 'Sally', 'age': 18, 'height': '187cm'}
```

Python - Dictionaries

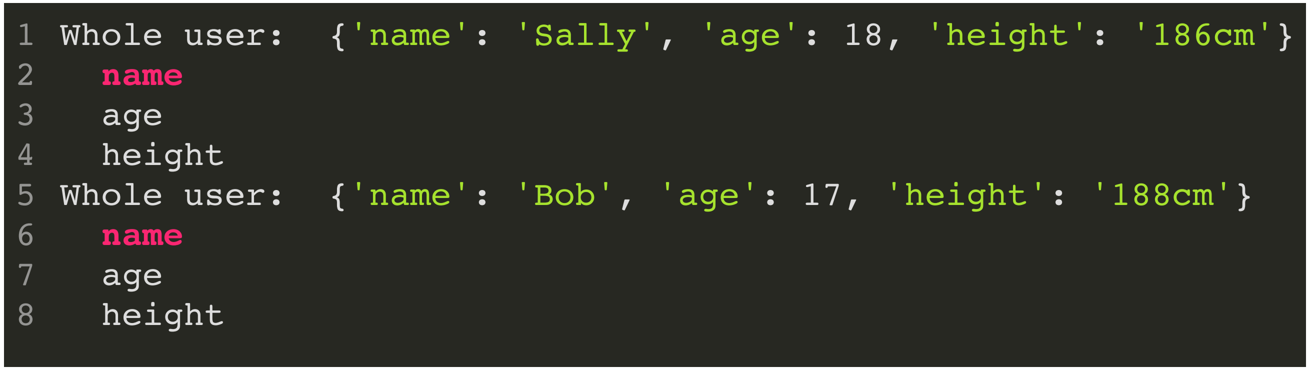
dict_loop.py

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

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



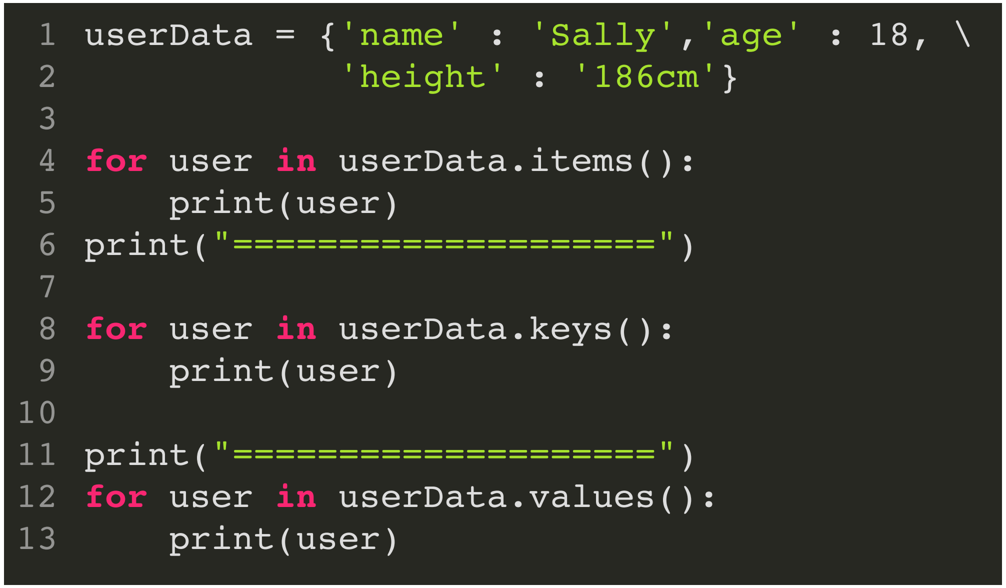
```python
userData = [
    {
        'name' : 'Sally',
        'age' : 18,
        'height' : '186cm',
    }, {
        'name' : 'Bob',
        'age' : 17,
        'height' : '188cm',
    },
]
for user in userData:
    print("Whole user: ", user)
    for part in user:
        print(f"  {part}")
```



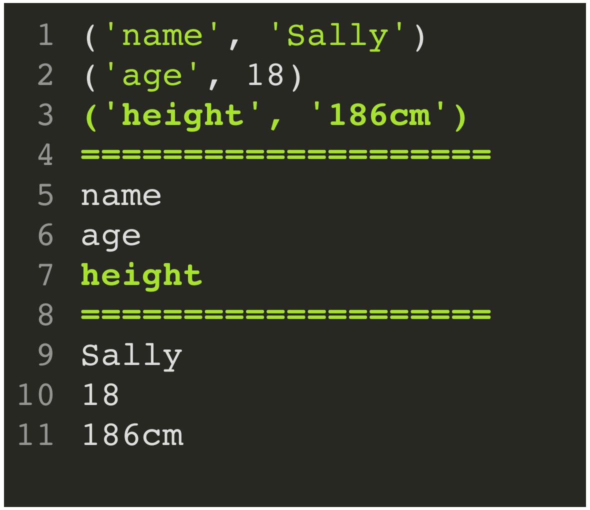
```
Whole user:  {'name': 'Sally', 'age': 18, 'height': '186cm'}
  name
  age
  height
Whole user:  {'name': 'Bob', 'age': 17, 'height': '188cm'}
  name
  age
  height
```

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

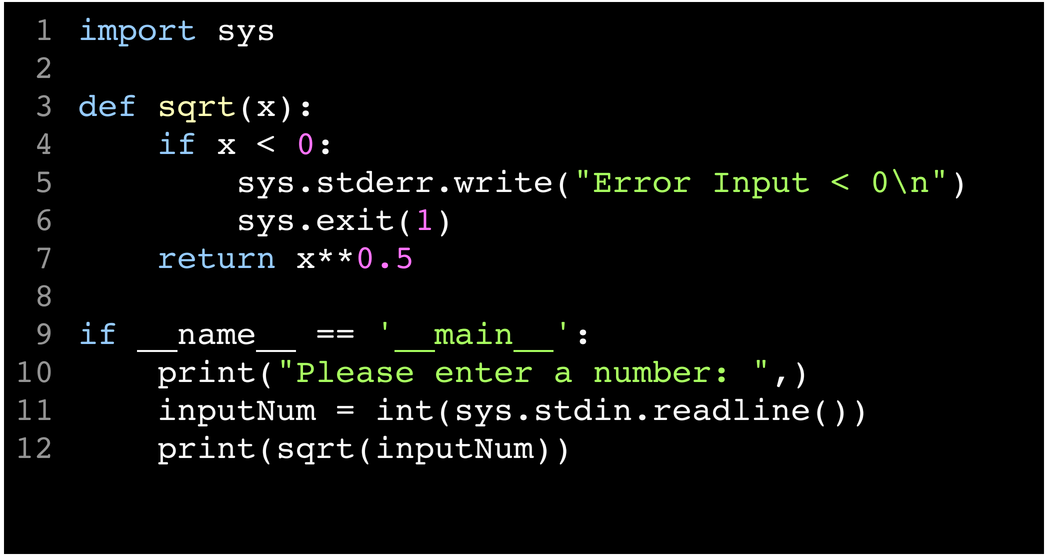
An **exception** is an action that disrupts the normal flow of a program. This action is often representative of an error being thrown. Exceptions are ways that we can elegantly recover from errors

Python - Exceptions

The simplest way to deal with problems...

Just crash

exception_1.py



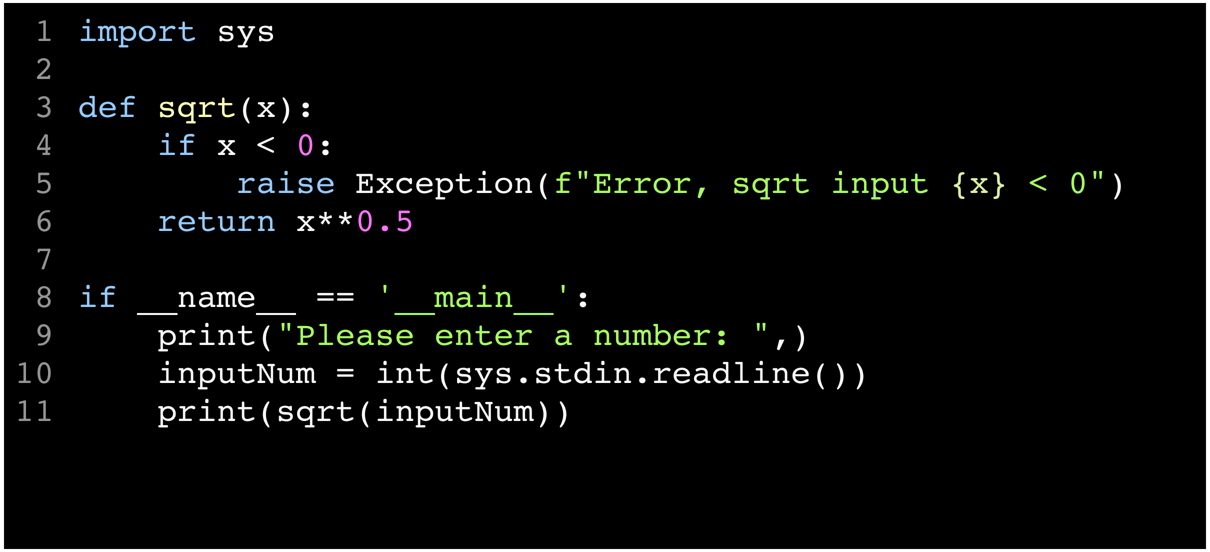
```python
import sys

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

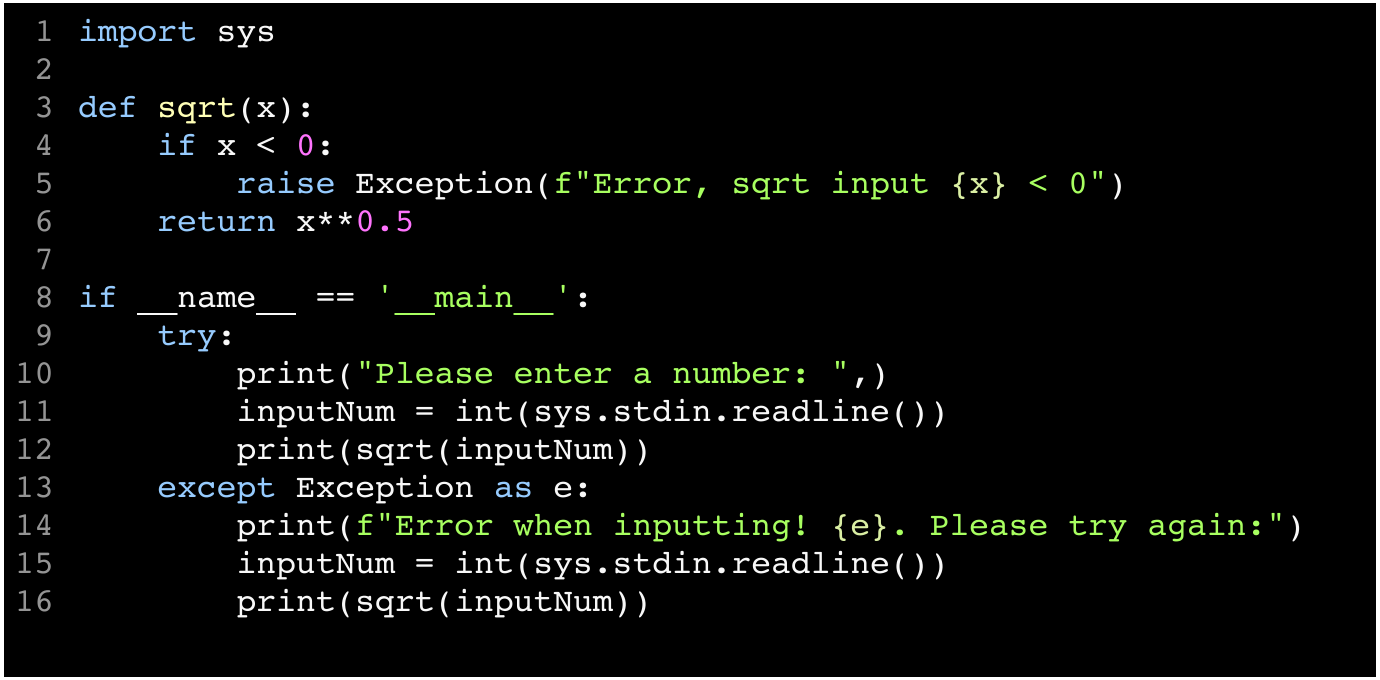


```python
import sys

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

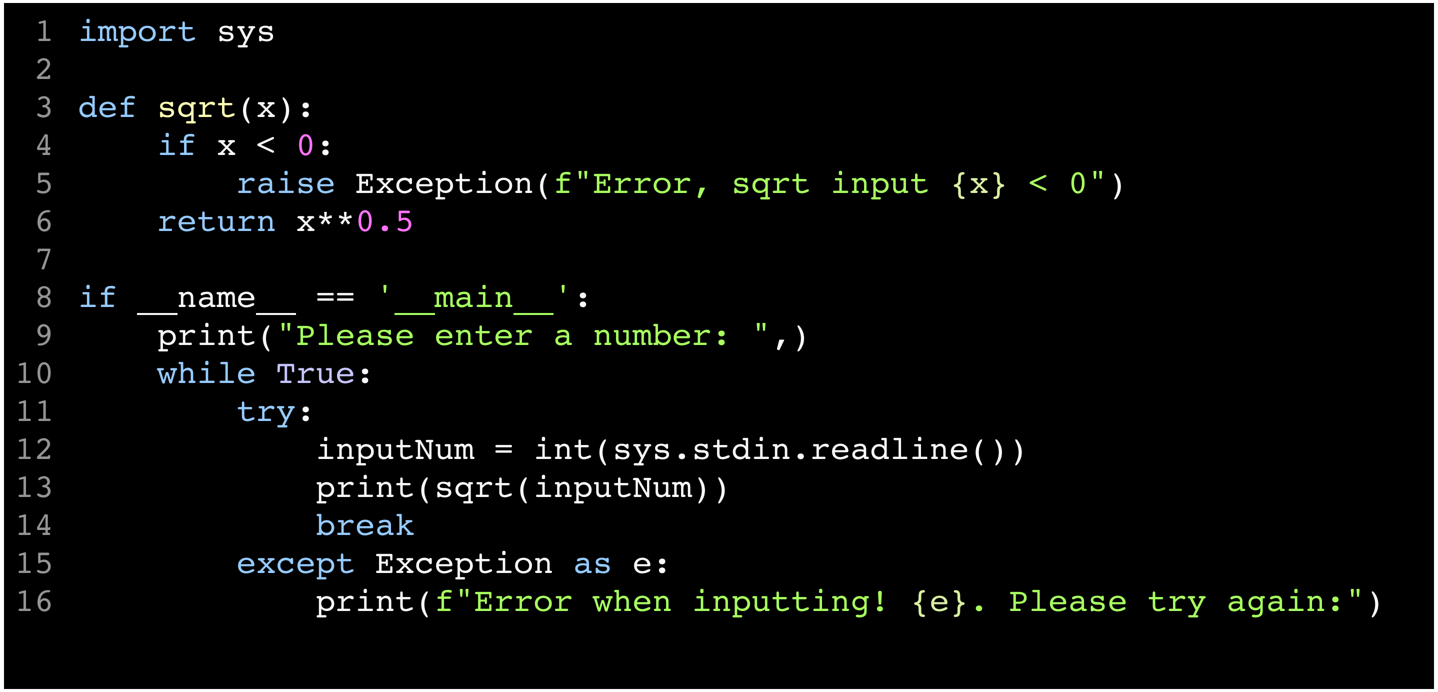


```python
import sys

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



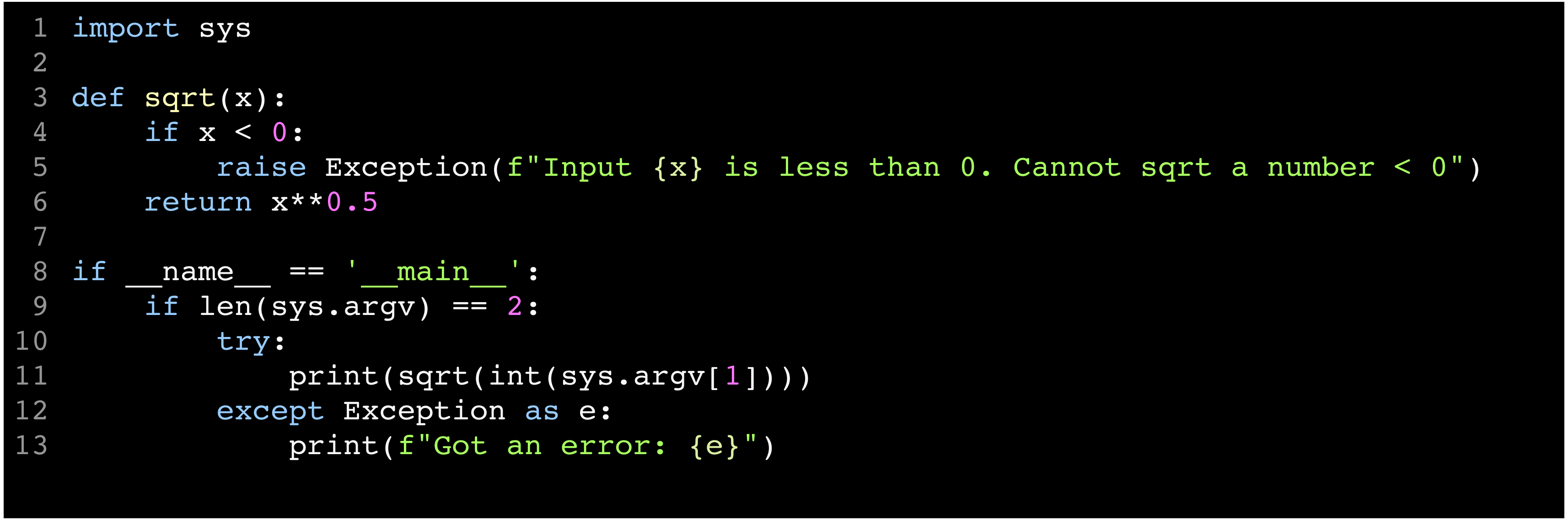
```python
import sys

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

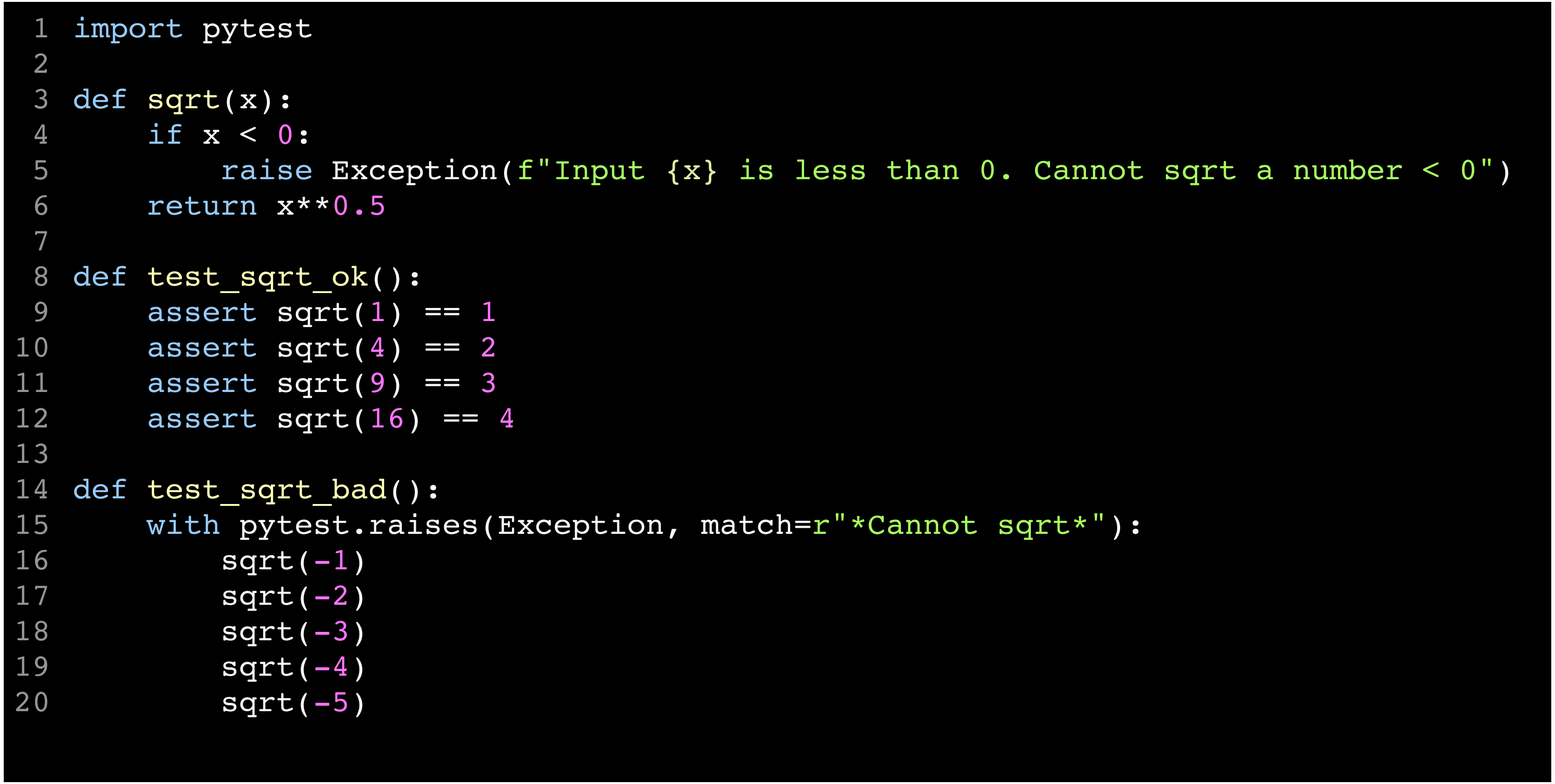


```python
import sys

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

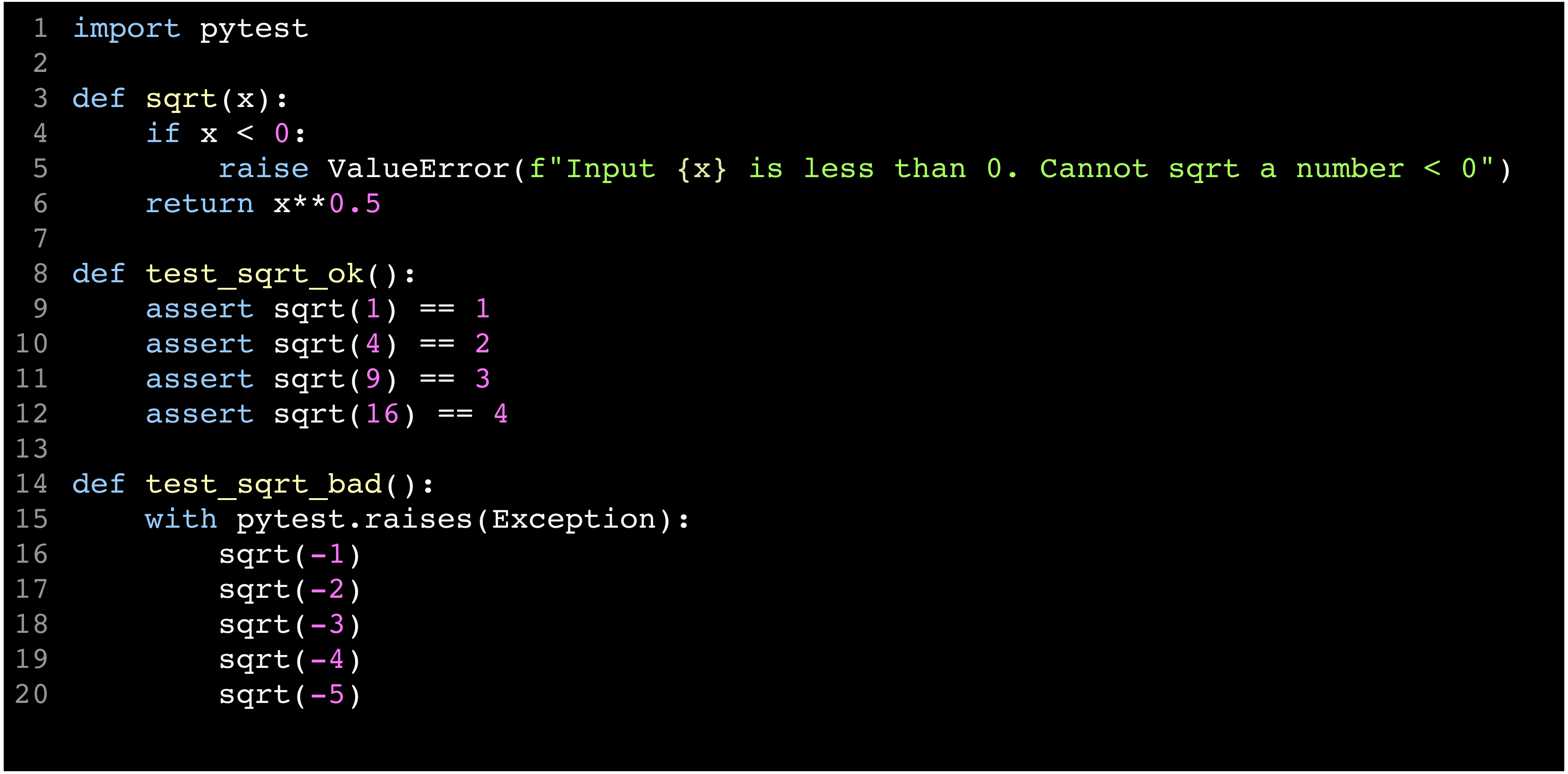


```python
import pytest

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



```python
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)
```