Foundations of Computing

Exceptions and Assertions; Iteration and Itertools

Ekaterina Vylomova
Summer Term 2023

THE UNIVERSITY OF MELBOURNE

February, 8th: On this day...(random facts)

- **1238** – The Mongols burn the Russian city of Vladimir.
- **1250** – Seventh Crusade: Crusaders engage Ayyubid forces in the Battle of Al Mansurah.
- **1879** – Sandford Fleming first proposes adoption of Universal Standard Time at a meeting of the Royal Canadian Institute.
- **1885** – The first government-approved Japanese immigrants arrive in Hawaii.
- **1946** – The People's Republic of Korea is dissolved in the North, establishing the communist-controlled Provisional People's Committee of North Korea.
- **1971** – The NASDAQ stock market index opens for the first time.

Announcements

- Worksheets 10–13 are due Feb, 10 (Fri) 11:59pm
- Project 1 is due Feb, 10 11:59pm
- Project 2 is on Grok!

Lecture Agenda

- Last lecture:
  - Mid-Term test: we started marking! Late submissions: we will only look at your latest submission before the end of the test.
  - Object-Oriented Programming
- This lecture:
  - Exception handling
  - Assertions
  - Iteration
  - Itertools

Lecture Outline

Exception Handling I

- Python prints the `Exception` causing the error:

```
>>> 9/0
Traceback (most recent call last):
  File "<web session>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero

>>> 1 + "2"
Traceback (most recent call last):
  File "<web session>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> 1 + i
Traceback (most recent call last):
  File "<web session>", line 1, in <module>
NameError: name 'i' is not defined
```

Exception Handling II

- Other common run-time exceptions are:
  - `IndexError`: raised when an index is out of range
  - `KeyError`: raised when a key is not found in a dictionary
- It is possible to "handle" exceptions within your code using `try: ... except Exception: ....`
- `try` attempts to execute its block of code, and passes off to the exception handlers (which are also tested in linear order) only if an exception is raised during the execution, before running the code block attached to `finally`

Exception Handling III

For example (does not work as expected):

```
x = "not a number"
while type(x) != int:
    x = int(input("Please enter a number: "))
```

Exception Handling IV

For example (does not work as expected):

```
x = "not a number"
while type(x) != int:
    x = int(input("Please enter a number: "))
```

Fixed:

```
x = "not a number"
while type(x) != int:
    try:
        x = int(input("Please enter a number: "))
    except ValueError:
        print("Oops! Try again...")
```

Exception Handling V

- You can catch/handle more than one exception

```
"""Print recipricol of input integer."""
while True:
    try:
        x = input("Please enter an integer: ")
        print(1/int(x))
    except ValueError:
        print("That's not an integer.")
    except ZeroDivisionError:
        print("Cannot find recipricol of 0")
```

Exception Handling VI

- It is considered best practice to only catch errors that you are interested in.

```
while True:
    try:
        x = input("Please enter an integer: ")
    except:
        print("Something went wrong... who knows what?")
```

- `try` and `except` should be used for 'exceptional' circumstances.

Lecture Outline

Assertions I

- To date, we have tended to assume well-behaved inputs to our functions etc., and lived with the fact that ill-behaved inputs will cause a logic or run-time error, e.g.:

```
def withdraw(amount,balance):
    if balance < -100:
        print("Insufficient balance")
        return(balance)
    else:
        print("Withdrawn")
        return(balance - amount)
>>> withdraw(100,False)
Withdrawn
-100
```

Assertions II

- One way to ensure that the inputs are of the right type is with assert:

```
def withdraw(amount,balance):
    assert type(balance) == int
    if balance < -100:
        print("Insufficient balance")
        return(balance)
    else:
        print("Withdrawn")
        return(balance - amount)
>>> withdraw(100,'a')
Traceback (most recent call last):
...
AssertionError
```

Assertions III

- Note, however, that assertions should be used sparingly and reserved for "impossible" code states
- Use an explicit `if` statement if the result is important to the logic of the code
- Assertions are an important tool in defensive programming

For vs While? I

What is the difference between `for..in..` and `while`?

- Imagine you could not use `for ... in ...:`
- How would you do this?

```
for element in [1, 2, 3]:
    print(element)
```

For vs While? II

What is the difference between `for..in..` and `while`?

- Imagine you could not use `for ... in ...:`
- How would you do this?

```
for element in [1, 2, 3]:
    print(element)
```

- Something like...

```
iterable = [1,2,3]
current_index = 0
while current_index < len(iterable):
    print(iterable[current_index])
    current_index += 1
```

What do you need to keep track of?

```
iterable = [1,2,3]
current_index = 0
while current_index < len(iterable):
    print(iterable[current_index])
    current_index += 1
```

- The iterable
- The current index in the iterable
- The end of the iterable

Python has Iterators to do this for you.

Lecture Outline

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

object	something you can manipulate
traverse	walk through/across
container	an object representing a collection of other objects (eg list, set, etc)

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

object	something you can manipulate
traverse	walk through/across
container	an object representing a collection of other objects (eg list, set, etc)

- Definition: an iterable object will return an iterator object when you pass it to the built-in Python function `iter()`.
(This happens automatically with `for..in..:`)

Iterator Objects I

- Iterators have a `__next__` method that will return the next thing in the iteration, and update their state/memory of where they are up to. You can access it with the built-in function `next()`
- Iterators raises a `StopIteration` exception when the container is empty

Iterator Objects II

- With explicit indexing:

```
iterable = [1,2,3]
current_index = 0
while current_index < len(iterable):
    print(iterable[current_index])
    current_index += 1
```

- With an iterator:

Iterator Objects III

```
iterator = iter([1,2,3])
while True:
    try:
        print(next(iterator))
    except StopIteration:
        break
```

Why? I

- There is less scope for a programmer to make an error
  - You do not have to initialise the index variable
  - You do not have to define the end value of the index variable
  - You do not have to increment the index variable
- Simply create the Iterator using `iter` and step through it using `next`. Python will take care of all the indexing.
- (Similarly, prefer `for` over `while` for a subset of those reasons.)

Why? II

- Iterators are more memory efficient than storing and indexing a whole iterable.
  Compare

```
FILENAME = 'jabberwocky.txt'
text = open(FILENAME).readlines()
for line in text:
    ...
```

  with

Why? III

```
FILENAME = 'jabberwocky.txt'
text = open(FILENAME)
for line in text:
    ...
```

Iterable vs. Iterator

- *Iterable* describes something that could conceptually be accessed element-by-element
- *Iterator* is an actual interface (or an object implementing the interface) allowing element-by-element access to its contents
- We use an iterator to access an iterable object

Iterators vs. Sequences

Iterators

- no random access
- "remembers" last item seen
- no `len()`
- can be infinite
- traverse exactly once (forwards)

Sequences

- supports random access
- doesn't track last item
- has `len()`
- must be finite
- "traverse" multiple times (fwd/rev/mix)

Lecture Outline

The `itertools` Module

- Implements a number of iterator "building blocks"
- Inspired by other programming languages (APL, Haskell, SML)
- Standardises a set of fast, memory efficient tools
- Each tool can be used alone or in combination
- Forms an "iterator algebra"

product: Cross-product of Sequences

```
from itertools import product
def get_deck():
    """Create a list of 52 cards."""
    suits = 'CDHS'
    values = '234567890JQKA'
    deck = product(values, suits)
    return (['' .join(c) for c in deck])
```

cycle: Repeating Items Indefinitely

```
from itertools import cycle
def deal():
    """Put cards in 4 equal piles."""
    deck = get_deck()
    hands = [[], [], [], []]
    players = cycle(hands)
    for card in deck:
        player = next(players)
        player.append(card)
    return(hands)
print(deal()[0])
print(deal()[1])
print(deal()[2])
print(deal()[3])
```

Example Uses

```
>>> deck = get_deck()
>>> len(deck)
52
>>> deck[:7]
['3C', '3S', '3D', '3H', '4C', '4S', '4D']
>>> hands = deal()
>>> hand1 = sorted(hands[0])
>>> hand1
['2C', '3H', '3S', '4C', '4S', '5S', '6D',
 '7D', '8D', 'AC', 'AD', 'AH', 'AS']
```

groupby: Group Items by Some Criterion

```
>>> from itertools import groupby
>>> def first(x): return(x[0])
>>> for rank, group in groupby(hand, first):
...   print(f"{rank} {list(group)}")
2 ['2C']
3 ['3H', '3S']
4 ['4C', '4S']
5 ['5S']
6 ['6D']
7 ['7D']
8 ['8D']
A ['AC','AD','AH', 'AS']
```

combinations: *n* Choose *k*

```
>>> from itertools import combinations
>>> aces = ['AC','AD','AH', 'AS']
>>> combinations(aces,2)
<itertools.combinations object at 0x1794998>
>>> list(combinations(aces,2))
[('AC', 'AD'), ('AC', 'AH'), ('AC', 'AS'),
 ('AD', 'AH'), ('AD', 'AS'), ('AH', 'AS')]
```

Lecture Summary

- How can we handle runtime errors gracefully?
- Exceptions: ask forgiveness not permission!
- How can we ensure that certain conditions are met?
- Assertions: raise an error if things are not as they should be.
- What is an iterator?
- Why are iterators useful?
- Differences between sequences and iterators
- The `itertools` module