Chapter 4 - Control Structures and Functions

CS 171 - Computer Programming 1
Lanzhou University
杨裔
18919801127
yy@lzu.edu.cn

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Outline

Control Structures

Python provides:

- Conditional branching
  - `if` statements
- Looping
  - `while` statements
  - `for ... in` statements

Conditional Branching

- General Syntax

```
if _boolean_expression1_:
    _suite1_
elif _boolean_expression2_:
    _suite2_
...
elif _boolean_expressionN_: _suiteN_
else:
    _else_suite_
```

- There can be zero or more `elif` clauses
- The final `else` clause is optional

Conditional Branching

- A (simple) `if ... else` statement can be expressed in a single conditional expression

  ```
  _expression1_ if _boolean_expression_ else _expression2_
  ```

- If `_boolean_expression_` evaluates to `True`,
  - the result is `_expression1_`
- Otherwise,
  - the result is `_expression2_`

Conditional Branching

```
offset = 20
if not sys.platform.startswith("win"):
    # sys.platform holds the name of the current platform,
    # for example, "win32" or "linux2"
    offset = 10

#same as above
offset = 20 if sys.platform.startswith("win") else 10
```

- We need to care when or not to use parenthesis;
- If we want to set `width` to 100 plus an extra 10 if `margin` is `True`:

```
width = 100 + (10 if margin else 0) # RIGHT!

width = 100 + 10 if margin else 0   # WRONG!

width = (100 + 10) if margin else 0 # Equivalent to the above!
```

Conditional Branching

What does this program do?

- You may want to try it with the values 0, 1 and 2 for count;

```
print("{0} file{1}".format(
                      (count if count != 0 else "no"),
                      ("s" if count != 1 else "")))
```

while Loops

- General Syntax

```
while _boolean_expression_:
    _while_suite_
else:
    _else_suite_
```

- The `else` clause is optional
- While `_boolean_expression_` is `True`, `_while_suite_` is executed
    - If a `continue` statement is found, control is immediately returned to the top of the loop
    - and `_boolean_expression_` is evaluated again
- When `_boolean_expression_` is `False`, `_else_suite_` is executed
    - if the `else` clause is present
    - and only once

while Loops

- General Syntax

```
while _boolean_expression_:
    _while_suite_
else:
    _else_suite_
```

- If the loop is broken due to a `break` or `return` statement
  - the `else` clause's suite is **not** executed

while Loops

- We can use while to search for an element in a list
- Such that if an element is not found, −1 is returned
  - Note: if you use list.index(), a ValueError exception is raised when an element is not found

```
def list_find(lst, target):
    index = 0
    while index < len(lst):
                          # we iterate over all elements of the list
        if lst[index] == target:
            break         # if we find target, we get out of the loop
        index += 1        # target is not found in the current index
    else:
        index = -1        # if target is not found in the list
    return index          # index will either have an index or -1
```

for Loops

- General Syntax (similar to `while`)

```
for _expression_ in _iterable_:
    _for_suite_
else:
    _else_suite_
```

- `expression` is either a single or a sequence of variable(s)
  - in the latter case, normally a tuple
- If `continue` is found, control is passed to the top of the loop
- If the loop runs to completion, it terminates
  - if present, the `else` suite is executed
- If the loop is broken out, due to `break`, `return` or an exception
  - the `else` clause's suite is **not** executed

for Loops

- The same function as before, but now using a `for` loop

```
def list_find(lst, target):
    for index, x in enumerate(lst):
        if x == target:
            break
        else:
            index = -1
        return index
```

- We are using `enumerate` to build an `iterable`

```
>>> for index, x in enumerate([’a’, ’b’, ’c’]): print(index, x);
...
0 a
1 b
2 c
```

Outline

Catching and Raising Exceptions

- General Syntax

```
try:
    _try_suite_
except _exception_group1_ as variable1:
    _except_suite1_
....
except _exception_groupN_ as _variableN_:
    _except_suiteN_
else:
    _else_suite_
finally:
    _finally_suite_
```

- There must be at least one `except` block
- Both `else` and `finally` are optional

Catching and Raising Exceptions

- General Syntax

```
try:
    _try_suite_
except _exception_group1_ as variable1:
    _except_suite1_
....
except _exception_groupN_ as _variableN_:
    _except_suiteN_
else:
    _else_suite_
finally:
    _finally_suite_
```

- `_else_suite_` is executed when `_try_suite_` finishes normally
    - but it is not executed if an exception occurs
- If a `finally` block exists, it is always executed at the end

Catching and Raising Exceptions

- General Syntax

```
try:
    _try_suite_
except _exception_group1_ as variable1:
    _except_suite1_
....
except _exception_groupN_ as _variableN_:
    _except_suiteN_
else:
    _else_suite_
finally:
    _finally_suite_
```

- Each `exception_group` can be a single exception or a tuple of exceptions
- For each group, the `as variable` part is optional

Catching and Raising Exceptions

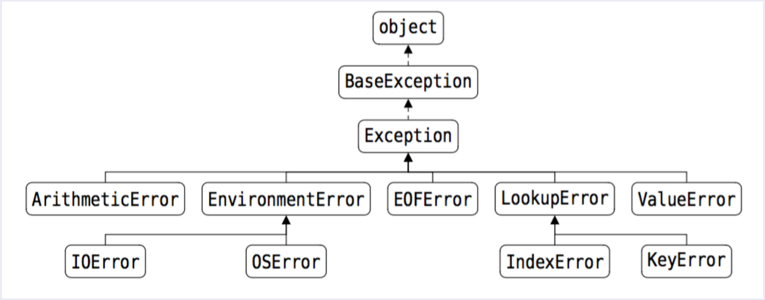
- General Syntax

```
try:
    _try_suite_
except _exception_group1_ as variable1:
    _except_suite1_
....
except _exception_groupN_ as _variableN_:
    _except_suiteN_
else:
    _else_suite_
finally:
    _finally_suite_
```

- If an exception occurs in `_try_suite_`, each except clause is tried
  - But, at most, only one is executed
  - If the exception matches an `exception_group`, the corresponding suite is executed

Catching and Raising Exceptions

- Part of Python's exception hierarchy:



- For reasons that will later be explained, you should always use the most specific exception types in an `exception_group`

Catching and Raising Exceptions

- Final version of `list_find()`, using exception-handling:

```
def list_find(lst, target):
    try:
        index = lst.index(target)
            # lst.index will raise a ValueError exception
            # if target is not found in the list
    except ValueError:
            # if a ValueError exception is raised,
            # the following statement will be executed
        index = -1
            # at this moment, index either has a valid value
            # or -1, as before
    return index
```

Catching and Raising Exceptions

- What does this function do?

```
def read_data(filename):
    lines = []
    fh = None
    try:
        fh = open(filename, encoding="utf8")
        for line in fh:
            if line.strip():
                lines.append(line)
    except (IOError, OSError) as err:
        print(err)
        return []
    finally:
        if fh is not None:
            fh.close()
return lines
```

Raising Exceptions

- We can use built-in exceptions, or create our own;
- General Syntax[a]:

```
raise _exception_(_args_)
```

- The `_exception_` should be either built-in or a custom exception derived from `Exception`
- If we give some text in `_args_`, this text will be output if the exception is printed when caught

[a] Other syntaxes are actually possible, but we will focus on this for now.

Custom Exceptions

- General Syntax:

```
class _exceptionName_(_baseException_): pass
```

- The `_base_` class should be `Exception` or a class that inherits from it

Custom Exceptions

- If we have a table object that holds rows, which hold columns, which have multiple items, we can search for a particular item:

```
found = False
for row, record in enumerate(table):
    for column, field in enumerate(record):
        for index, item in enumerate(field):
            if item == target:
                found = True
                break
        if found:
            break
    if found:
        break
if found:
    print("found at ({0}, {1}, {2})".format(row, column, index))
else:
    print("not found")
```

- This code is complicated: we need to break each loop separately

Custom Exceptions

- An alternative solution is to use a custom exception:

```
class FoundException(Exception): pass

try:
    for row, record in enumerate(table):
        for column, field in enumerate(record):
            for index, item in enumerate(field):
                if item == target:
                    # if the value is found, an exception is raised
                    raise FoundException()
except FoundException:
                    # the control of the program then comes here
    print("found at ({0}, {1}, {2})".format(row, column, index))
else:
                    # if not exception is raised, this is executed:
    print("not found")
```

Custom Exceptions

- What does this program do?

```
class Error(Exception): pass
class ValueTooSmallError(Error): pass
class ValueTooLargeError(Error): pass

number = 10

while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:   raise ValueTooSmallError
        elif i_num > number: raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
    except ValueTooLargeError:
        print("This value is too large, try again!")

print("Congratulations! You guessed it correctly.")
```

Outline

Custom Functions

- General Syntax:

```
def _functionName_(_parameters_):
    _suite_
```

- The `_parameters_` are optional
  - if there is more than one, they are separated by commas
  - or a sequence of `identifier = value` pairs
- To calculate the area of a triangle using Heron's formula:

```
def heron(a, b, c):
    s = (a + b + c) / 2
    return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

Custom Functions

- To calculate the area of a triangle using Heron's formula:

```
def heron(a, b, c):
    s = (a + b + c) / 2
    return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

- Inside the function, each parameter `a`, `b` and `c` is initialized with the corresponding value that was passed as an argument;
- When the function is called, we must supply all of the arguments
  - for example, `heron(3,4,5)`
    - in this case, `a` is set to 3, `b` is set to 4 and `c` is set to 5
  - if we give too few/too many arguments, a `TypeError` exception occurs

Custom Functions

- Every function in Python returns a value
- But it is acceptable to ignore the return value of a function;
- The return value may be a single value, or a collection of values
- We can leave a function at any point by using the `return` statement
- If we `return` with no arguments, or if we don't return, the function returns `None`

Custom Functions

- Some functions have parameters that assume default values:

```
def letter_count(text, letters=string.ascii_letters):
    letters = frozenset(letters)
    count = 0
    for char in text:
        if char in letters:
            count += 1
    return count
```

- `letter_count` can then be called with just one argument:
  - for example, `letter_count("Maggie and Hopey")`
    - in this case, `text` is set to `"Maggie and Hopey"`,
    - and `letters` is set to `string.ascii_letters`, i.e., to `'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`
- `letter_count` can also be called with two arguments, changing the default value:
  - for example, `letter_count("Maggie and Hopey", "aeiouAEIOU")`
  - or `letter_count("Maggie and Hopey", letters="aeiouAEIOU")`

Custom Functions

- If we want to use default mutable arguments, we can:

```
def append_if_even(x, lst=None):
    if lst is None:
        lst = []
    if x % 2 == 0:
        lst.append(x)
    return lst
```

Function naming

- You should use a consistent naming scheme; we have been using:
  - `UPPERCASE` for constants
  - `TitleCase` for classes (including exceptions)
  - `lowercase` or `lowercase_with_underscores` for everything else
- Avoid abbreviations,
  - unless they are standardized (e.g., using `i` for a loop counter);
- Variable and parameter names should be long enough to be descriptive
  - the name should describe the meaning of the data, and not its type
    - use, e.g., `amount_due` instead of `money`
  - Functions and methods should have names that say what they *do* or *return*, but not how they do it (since that might change)

```
def find(l, s, i=0):                                 # BAD
def linear_search(l, s, i=0):                        # BAD
def first_index_of(sorted_name_list, name, start=0): # GOOD
```

Docstrings

- We can add documentation to any function using a *docstring*
  - this is a string that comes immediately after the def line

```
def shorten(text, length=25, indicator="..."):
    """Returns text or a truncated copy with the indicator added

    text is any string; length is the maximum length of the returned
    string (including any indicator); indicator is the string added
    at the end to indicate that the text has been shortened

    >>> shorten("Second Variety")
    'Second Variety'
    >>> shorten("Voices from the Street", 17)
    'Voices from th...'
    >>> shorten("Radio Free Albemuth", 10, "*")
    'Radio Fre*'
    """
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text
```

Argument and Parameter Unpacking

- If we have a list, named `sides`, of 3 integers
  - we can call `heron(sides[0], sides[1], sides[2])`
- Alternatively, we can use the sequence unpacking operator `*` to supply positional arguments
  - we can call `heron(*sides)`
- If the list has more items that the function has parameters
  - we can use slicing to extract the right number of arguments

Argument and Parameter Unpacking

- We can use the sequence unpacking operator * in a function's parameter list
  - this is useful to create functions that can take a variable number of positional arguments

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result

product(1, 2, 3, 4)  # args == (1, 2, 3, 4); returns: 24
product(5, 3, 8)     # args == (5, 3, 8); returns: 120
product(11)          # args == (11,); returns: 11
```

Argument and Parameter Unpacking

- We can also have keyword arguments following positional arguments:

```
def sum_of_powers(*args, power=1):
    result = 0
    for arg in args:
        result += arg ** power
    return result
```

- What does this function do?
- It can be called with just positional arguments
    - for example, `sum_of_powers(1, 3, 5)`
- It can be also called with both positional and keyword arguments
    - for example, `sum_of_powers(1, 3, 5, power = 2)`