Control Structures

Structured Programming

Any algorithm can be expressed by:

- Sequence - one statement after another
- Selection - conditional execution (not conditional jumping)
- Repetition - loops

We've already seen sequences of statements. Today we'll learn selection (conditional execution), and repetition.

The `if-else` Statement

Conditional execution:

```
if boolean_expression:
    # a single statement executed when boolean_expression is true
else:
    # a single statement executed when boolean_expression is false
```

- `boolean_expression` is not enclosed in parentheses
- `else:` not required

Example:

```
if (num % 2) == 0:
    print("I like " + str(num))
else:
    print("I'm ambivalent about " + str(num))
```

Blocks

Python is block-structured. Contiguous sequences of statements at the same indentation level form a block. Blocks are like single statements (not expressions - they don't have values).

```
if num % 2 == 0:
   print(str(num) + " is even.")
   print("I like even numbers.")
else:
    print(str(num) + " is odd.");
    print("I'm ambivalent about odd numbers.")
```

Multi-way `if-else` Statements

This is hard to follow:

```
if color == "red":
    print("Redrum!")
else:
    if color == "yellow":
        print("Submarine")
    else:
        print("A Lack of Color")
```

This multi-way if-else is equivalent, and clearer:

```
if color == "red":
    print("Redrum!")
elif color == "yellow":
    print("Submarine")
else:
    print("A Lack of Color")
```

`if`-`else` Expression

One often wants to assign different values to a variable based on a condition. `if-else` expressions can reduce some of the verbosity of `if-else` statements.

```
>>> def current_conditions():
...     return {'precipitation': None}
...
>>> is_raining = current_conditions()['precipitation']
>>> cooking_method = "grilling" if not is_raining else "baking"
>>> cooking_method
'grilling'
>>>
```

Shake & Bake!

What is the value of the expression `"result" if not "first" else "last"`?

Loops

Algorithms often call for repeated action, e.g. :

- "repeat ... while (or until) some condition is true" (looping) or
- "for each element of this array/list/etc. ..." (iteration)

Python provides two control structures for repeated actions:

- `while` loop
- `for` iteration statement

while Loops

while loops are pre-test loops: the loop condition is tested before the loop body is executed

```
while condition: # condition is any boolean expression
    # loop body executes as long as condition is true
```

Example

```
>>> def countdown(n):
...     while n > 0:
...         print(n)
...         n -= 1
...
print('Blast off!')
...
>>> countdown(5)
5
4
3
2
1
Blast off!
```

`for` Statements

`for` is an iteration statement

- iteration means visiting the elements of an iterable data structure

In the for loop:

```
>>> animal = 'Peacock'
>>> for animal in ['Giraffe', 'Alligator', 'Liger']:
...     print(animal)
...
Giraffe
Alligator
Liger
>>> animal
'Liger'
```

- `animal` is assigned to each element of the iterable list of animals in successive executions of the `for` loop's body
- notice that the loop variable re-assigned an existing variable

break and else

- break terminates execution of a loop
- optional else clause executes only of loop completes without executing a break

```
>>> def sweet_animals(animals):
...     for animal in animals:
...         print(animal)
...         if animal == 'Liger':
...             print('Mad drawing skillz!')
...             break
...     else:
...         print('No animals of note.')
...
>>> sweet_animals(['Peacock', 'Liger', 'Alligator'])
Peacock
Liger
Mad drawing skillz!
>>> sweet_animals(['Peacock', 'Tiger', 'Alligator'])
Peacock
Tiger
Alligator
No animals of note.
```

Run-time Errors

An error detected during execution is called an exception and is represented at runtime by an exception object. The Python interpreter raises an exception at the point an error occurs. The exception is handled by some exception-handling code. Here we don't handle the ValueError ourselves, so it's handled by the Python shell:

```
>>> int('e')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'e'
```

We can handle an exception by enclosing potentially error-raising code in a try block and handling errors in an except clause.

```
try:
    code_that_may_raise_error()
except ExceptionType as e:
    print(str(e))
    code_that_handles_exception()
```

`ExceptionType` and `as e` are optional. If left off, except clause will catch any exception.

Exception Handling Example

```
>>> def get_number_from_user():
...     input_is_invalid = True
...     while input_is_invalid:
...         num = input('Please enter a whole number: ')
...         try:
...             num = int(num)
...             # Won't get here if exception is raised. '
...             input_is_invalid = False
...         except ValueError:
...             print(num + ' is not a whole number. Try again.')
...     return num
...
>>> get_number_from_user()
Please enter a whole number: e
e is not a whole number. Try again.
Please enter a whole number: 3
3
```

Raising Exceptions

Figure: `raise UnsafeBabyError('Not secured to motorcycle')`

You can use exceptions for error handling in your code by raising exceptions.

Super Troopers

Here's a snippet that ensures construction of a valid `SuperTrooper`:

```
class SuperTrooper(Trooper):
    job = 'Hilarity'

    def __init__(self, name, is_mustached):
        super().__init__(name)
        # Discovers the error:
        if not is_mustached:
            # Create an instance of an exception class and raise it:
            raise ValueError('A Super Trooper must have a mustache')
```

If you try to create a `SuperTrooper` without a mustache, you get a `ValueError`:

```
>>> import trooper
>>> trooper.SuperTrooper("Dr.CS", is_mustached=False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "[path elided]/trooper.py", line 25, in __init__
    raise ValueError('A Super Trooper must have a mustache')
ValueError: A Super Trooper must have a mustache
```

Exception Code Design

Error handling using exceptions involves:

1. The exception object that represents the error and contains information about the error
   - Exception objects must derive from `BaseException`. If you define your own exception, use `Exception` as the base class
   - Best to simply use exceptions already defined in the standard library
2. the code that discovers the error, creates the exception object and `raise` s it to be caught by the error handling code, and
3. the code that catches the exception and handles the error the exception represents

Creating exceptions is more common in library code. Applications will often catch exceptions, which is straightforward.

Exception Code Design Example

```
def get_number_from_user():
    is_valid = False
    while not is_valid:
        num = input('Please enter a whole number: ')
        try:
            # Input problem discovered and raised inside int function
            num = int(num)
            # Won't get here if exception is raised. '
            is_valid = True
        # Input value problem caught and handled in "catch clause"
        except ValueError:
            print(num + ' is not a whole number. Try again.')
   return num
```

For more details on exceptions, see
https://docs.python.org/3/tutorial/errors.html

Conclusion

Python provides all the control structures you need for controlling program flow:

- Sequence - one statement after another
- Selection
  - `if` - `elif` - `else` statements
  - `if` - `else` expressions
- Repetition
  - `while` loops
  - `for` iteration statements