COMP10001 Foundations of Computing
Functions, Methods and Iteration

Semester 2, 2018

Chris Leckie & Nic Geard

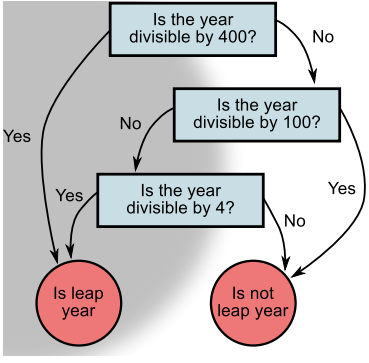
THE UNIVERSITY OF MELBOURNE

Lecture Agenda

- Last lecture:
  - `bool`
  - `if`
  - Conditionals
- This lecture:
  - Conditional Recap
  - Functions
  - Iteration

Conditional Recap

- Problem: evaluate whether a given year is a leap year (`True`) or not (`False`)
- Flowchart:



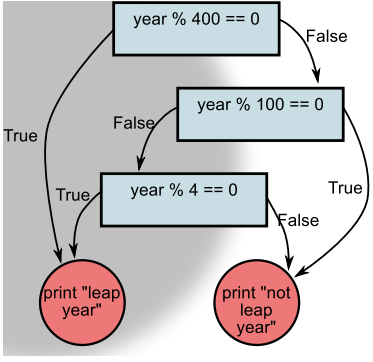
Cascading Conditions

- It is possible to test various **mutually-exclusive** conditions by adding extra conditions with `elif`, and possibly a catch-all final state with `else`

```
if year % 400 == 0:
    print("leap year")
elif year % 100 == 0:
    print("not leap year")
elif year % 4 == 0:
    print("leap year")
else:
    print("not leap year")
```

Conditional Recap

- Problem: evaluate whether a given year is a leap year (`True`) or not (`False`)
- Pythonic flowchart:



Class Exercise

- Simply the preceding code into one `if` statement and one `else` statement (and no `elif` statements)

Functions: Introduction

- What's a function?
  - (much like in Maths) functions take a set of input values, perform some calculation based on them, and optionally return a value
  - you have already seen and used a bunch of functions by this stage, e.g.: `ord()`, `range()`, `str()`, `len()`, ...
- Wouldn't it be nice to be able to recycle chunks of our own code?

Functions: The Details

- In order to define a function, we need:
  - A function name (following same conventions as other variable names)
  - (optionally) a list of input variables
  - (optionally) a UNIQUE output object (via `return`)
- Basic form:

```
def␣NAME(INPUTLIST):
␣␣␣␣statement␣block
```

Warm-up Functions

- Print the number of digits in a number

```
def print_digits(n):
    s = str(abs(n))
    print(len(s) - ('.' in s))
```

- Convert from Celsius to Fahrenheit:

```
def C2F(n):
    print(9*n/5 + 32)
```

Functions: More Details

- It is possible to define "variable-arity" functions (i.e. functions which take variable numbers of arguments) by specifying default values for arguments:

```
def seconds_in_year(days=365):
    return(days*24*60*60)
```

```
>>> seconds_in_year()
31536000
>>> seconds_in_year(366)
31622400
```

The Power of `return` I

- In order to use the output of a function (eg to assign it to a variable), we need to `return` a value:

```
def count_digits(n):
    s = str(abs(n))
    return(len(s) - ('.' in s))

print(count_digits(-123.123))
```

- Convert from Celsius to Fahrenheit:

```
def C2F(n):
    return(9*n/5 + 32)

print(C2F(21))
```

The Power of `return` II

- `return` is also a way of (unconditionally and irrevocably) terminating a function:

```
def safe_divide(x,y):
    if y:
        return(x/y)

    print("ERROR: denom must be non-zero")
```

Quiz

What is printed to the screen here?

```
def bloodify(word):
    return(word[:3] + '-bloody-' + word[3:])

w = bloodify('fantastic')
print(w)
```

What is printed to the screen here?

```
def bloodify(word):
    return(word[:3] + '-bloody-' + word[3:])

w = bloodify('andrew')
print(w)
```

Variables and "Scope" I

- Each function (call) defines its own local variable "scope". Its variables are not accessible from outside the function (call)

```
def subtract_one(k):
    k = k - 1
    return(k)

i = 0
n = subtract_one(i)
print(i)
print(n)
print(k)
```

Variables and "Scope" II

- Are the semantics different to the previous slide?

```
def subtract_one(i):
    i = i - 1
    return(i)

i = 0
n = subtract_one(i)
print(i)
print(n)
print(k)
```

Variables and "Scope" III

- Functions can access variables defined outside functions ("global" variables), although they should be used with extreme caution (perhaps never!)

```
def fun1(j):
    fun2(j)
    return(1)
def fun2(k):
    global i,j  # global variables
    print(i,j,k)
    return(2)
i,j,k = 1,2,3
fun1(i)
```

Reasons for Using Functions

- "Archiving" code in libraries
- Removing redundancy
- Ease of testing
- Increasing modularity
- Increasing readability

Iteration

One final, essential tool to make the computer do something over and over again.

- Repeat something forever (eg Windows)
- Repeat something a fixed number of times
  - move Mario forward 10 pixels
  - print 7 copies
  - play Nyan Cat 15 times
- Repeat something until something happens (eg scroll while button held)

Iteration: `while` Loops I

A conditional loop.

- The general idea is that we continue repeating a block of code as long as a given condition holds
- Basic form:

```
while␣<condition>:
␣␣␣␣statement␣block
```

- We use the notion of "block" as in `if` statements, but here, potentially the code block is repeated

```
text = ''
while len(text) != 3:
  text = input('Enter 3-digit code: ')
```

Iteration: `while` Loops II

- Another way to end `while` loops (and bypass the condition in the `while` statement) is via a `break` in the block of code

```
text = ''
while True:
  text = input('Enter 3-digit code: ')
  if len(text) == 3:
    break
  print('Sorry, invalid code.')
```

This prematurely and unconditionally exits from the loop

Iteration: `for` Loops I

A loop over sequences.

- The general idea is that we work our way through (all of) an "iterable" (eg `str`, `tuple`) of items one item at a time, in sequence
- Basic form:

```
for␣<var>␣in␣<iterable>:
␣␣␣␣statement␣block
```

- Note: `in` here is not (quite) the same as the comparison operator of the same name

Iteration: `for` Loops II

- Simple example:

```
sum = 0
for i in (1,2,3):
    sum = sum + i
print(sum)
```

is equivalent to:

```
sum = 0
sum = sum + 1
sum = sum + 2
sum = sum + 3
print(sum)
```

Iteration: `for` Loops III

- More interesting example:

```
vowels = 0
for char in "rhythm":
    if char in "aeiou":
        vowels = vowels + 1
print(vowels)
```

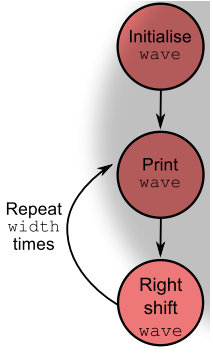
A Useful Function for creating a sequence

- `range(start=0,end,step=1)`: generate a sequence of `int` values from `start` (inclusive) to `end` (non-inclusive), counting `step` at a time

```
>>> for i in range(5):
...     print(i, end=" ")
0 1 2 3 4
>>> for i in range(0,10,2):
...     print(i, end=" ")
0 2 4 6 8
>>> for i in range(10,0,-1):
...     print(i, end=" ")
10 9 8 7 6 5 4 3 2 1
```

`for` Loop Practice: Mexican Wave

- Given the string `wave` made up of a `"Y"` and `width-1` repeats of `"x"`, how can we use a `for` loop to move the `"Y"` across one position to the right at a time?



Choosing between `for` and `while`

- If you need to iterate over all items of an iterable, use a `for` loop
- If there is a well defined end-point to the iteration which doesn't involve iterating over all items, use a `while` loop
- With a `for` loop, avoid modifying the object you are iterating over within the block of code
- Given a choice between the two, `for` loops are generally more elegant/safer/easier to understand

Class Exercise

- Assuming an unlimited number of coins of each of the following denominations:

  $$(1, 2, 5, 10, 20)$$

  calculate the number of distinct coin combinations which make up a given amount N (in cents).

Lecture Summary

- What is an "iteration", and what are the two basic forms of iteration in python?
- What is the basic syntax of a `for` loop, and what function is commonly used to iterate a fixed number of times?
- What is the basic syntax of the `while` loop, and what is the role of `break`?
- When should you use `for` vs. `while`?