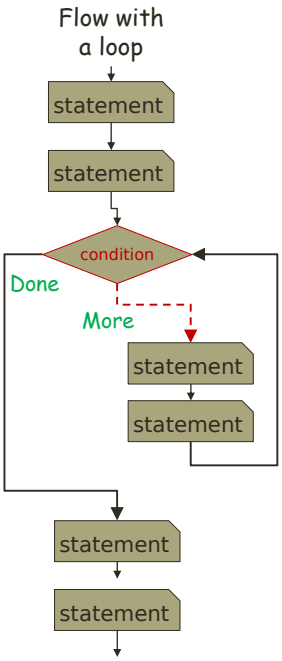
Loops

zyBooks
"Programming in Python 3"
chapter 5

Loops

- Loops produce behavior that is repeated
- Variable number of repetitions:
  - Number usually unknown beforehand
  - 0 or more times ("while" loop)
  - at least once ("repeat" loop)
    - *not in Python*
- Known number of repetitions
  - "for" loop



"*While*" *loops*

- Like an "if" statement with repetition
  - Evaluate an expression
  - If True, execute indented statements
  - Re-evaluate the expression
  - ...
- Indented statements need to modify the value of the test expression
  - If they don't, the loop won't ever end - "infinite loop", almost always an undesirable thing

Example: the Collatz Conjecture

- Hailstone sequence: series of x values generated by:
  - $x_n = \begin{cases} \frac{x_{n-1}}{2}, & \text{if } x_{n-1} \text{ even} \\ (3 \times x_{n-1}) + 1, & \text{if } x_{n-1} \text{ odd} \end{cases}$

  with arbitrary positive starting value x_0
- Conjecture: this series always reaches x_n=1, regardless of the starting value x_0
- Demonstrate: write a program to generate and print the hailstone sequence
- *(pause...)*

Hailstone Sequence Solution

```
#!/usr/bin/env python3
# Simple while loop:
#       the hailstone sequence

hailstone = int( input("Enter a positive integer: ") )

while hailstone != 1:
    if (hailstone % 2) == 0:
        hailstone = hailstone//2
    else:
        hailstone = 3*hailstone + 1
    print(hailstone, end='  ')

print('\nReached the end!')
```

- Can you:
  - Do this with a conditional expression?
  - Put the whole script in a while loop, to calculate multiple sequences?

Faking a Repeat Loop

- Guarantee at least one pass through the loop:
  - Use "always-true" test expression
  - Use "break" statement at end of loop to terminate
- Example:
  - ```
    count = 0
    while True:
        count += 1
        print('Loop', count)
        answer = input('again? ')
        if answer in ['no', 'No', 'NO']:
            break
    print('Done looping')
    ```

RETURN to "break Statement"
    ```

Counting Loops

- Used in various situations
- "Loop counter" variable *increments* each time around a loop
  - Increment can be additive, multiplicative, etc.
- Example: powers-of-two table

```
p = 1
while p <= 5000:
    print('%4d' % p)
    p *= 2
print('#-----')
```

- Loop counter doubles on each pass

Powers of 2, Version 2

```
#!/usr/bin/env python3

limit = int( input('How many powers? ') )

n = 0
print('#------------#')
while n <= limit:
    print(' %3d %8d' % (n, 2**n))
    n += 1
print('#------------#')
```

- Loop counter increases by 1 on each pass
- Common pattern

For loops

- Apply a block of code to every member of a *collection* of elements
  - Each character of a string
  - Every item in a list or tuple
  - Each key:value pair in a dictionary
- Generic approach: use a loop counter to access sequential positions in the collection
- Python approach: use a *for* loop that automatically accesses each member
  - Also known as "for-each" loops

Generic Approach

- Use the **`len()`** function with a loop counter:
  - ```
    i = 0
    while i < len( mystring ):
        print('char %i: %s' % (i, mystring[i]))
    ```
  - ```
    i = 0
    while i < len( mylist ):
        member = mylist[ i ]
        #
        # do something with the list member
        #
    ```
  - ```
    i = 0
    while i < len( mydict.keys() ):
        k = mydict.keys[i]
        print('key:', k, '  value:', mydict[k])
    ```
    ```

Python Approach

- Access each element with *for* loop:
  - ```
    for c in mystring:
        print('char %s' % c)
    ```
  - ```
    for member in mylist:
        #
        # do something with the list member
        #
    ```
  - ```
    for k in mydict.keys():
        print('key:', k, '  value:', mydict[k])
    ```

The range() Function

- Generates a sequence of *integers*
  - No decimal points allowed!
- Optional: starting value, "stop-at" value, increment
- Examples:
  - `r = range(10)` # yields 0 .. 9
  - `r = range(-5, 7)` # yields -5, -4, -3, .. 5, 6
  - `r = range(0, 100, 2)` # yields even numbers < 100
- Python 3: `range()` returns a "list-like" object that yields the sequence
  - Python 2: `range()` returns an actual list of numbers
    ```

Example: Downward-counting Range

```
In [11]: r = range(4, 0, -1)

In [12]: print(r)
range(4, 0, -1)

In [13]: rlist = []

In [14]: for i in r:
    ...:     rlist.append(i)
    ...: print(rlist)
    ...:
    ...:
[4, 3, 2, 1]

In [15]:
```

"list-like" object

Powers of 2, Version 3

```
#!/usr/bin/env python3

limit = int( input('How many powers? ') )

print('#------------#')
for n in range(limit+1):
    print(' %3d %8d' % (n, 2**n))
print('#------------#')
```

- Typical use of `range()` with a `for` loop
- Use "`range(limit+1)`" to include "`limit`" in the values

exercise

- build powers table
  - input maximum power
  - input *list of bases*
  - for each power, each base, print power

The break Statement

- Sometimes an exceptional condition requires a loop to terminate early
- Example: user input requests to quit
  - the "Faking a Repeat Loop" slide
- Can make a loop's logic easier to understand
- Example: counted loop that terminates early if a condition is met
  - Finding prime numbers with nested for loops *(next slide)*

Example: search text for a keyword

- Read in text from a file
  - Use readlines() instead of read()
- Split line into words
- Check for keyword in line
  - if found, break out of reading loop
- create list of words

Example: break in a Primes-Finder

```
#!/usr/bin/env python3
import pprint

limit = int(input('Largest value? '))

primes = [2,3]
for candidate in range(primes[-1], limit, 2):

    isPrime = True
    for p in primes:
        if candidate % p == 0:
            isPrime = False
            break
    if isPrime:
        primes.append(candidate)

pprint.pprint(primes)
```

The continue Statement

- Terminates the current pass through a loop
  - Next pass unaffected
  - Useful for "skipping" particular situations
  - Could also be done with "if-else" blocks
- Example – average length of fruit names:

```
fruit = ['banana', 'kiwi', 'walnut']
nuts = ['almond', 'pecan', 'walnut']
namelengths = []
for f in fruit:
    if f in nuts: # exclude nuts from counts
        continue
    namelens.append( len(f) )
print( sum(namelengths) / len(namelengths) )
```

Example 2: search text for a keyword

- Read in text from a file
  - Use readlines() instead of read()
- Split line into words
- Check for keyword in line
  - if not found, continue
- process line – create list of words in the line

Loops' else Clause

- Applies only if a loop terminates *normally*
  - *i.e.* if no `break` occurs
- Optional addition to `while` and `for` loops
- Not related to the "`else`" in an "`if-else`" statement
- Sample structure:
  - ```
    while <condition> :
        # do stuff …
        if <bad-condition> :
            break
    else:
        # do normal-end-of-loop stuff…
    ```

Primes-finder Revisited

```
#!/usr/bin/env python3
import pprint

limit = int(input('Largest value? '))

primes = [2,3]
for candidate in range(primes[-1], limit, 2):

    for p in primes:
        if candidate % p == 0:
            isPrime = False
            break
    else:
        primes.append(candidate)

pprint.pprint(primes)
```
```

The enumerate() Function

- For loops can loop over elements in a list or tuple, or over the indexes of the list/tuple
  - Use "`range( len(mylst) )`" to get "mylst" indexes
- `enumerate()` conveniently provides elements and element indexes together
- Example:

```
name = input('Your name? ')
for (index, letter) in enumerate(name):
    if letter.upper() in 'AEIOU':
        print('Found vowel', letter)
        print('  at position', index)
```