# Files

- Mostly handle like any sequential data type
  - A sequence of characters if a text file, or a sequence of bytes if a binary file
- First open file, and say purpose – read or write

```
inputFile = open('mydata.txt', 'r')
outputFile = open('myresults.txt', 'w')
```

- Often process text files as a sequence of lines

```
for line in inputFile: # process each line as a string
    outputFile.write(line[:5] + '\n')
```

- Best to close the files when you're done

```
inputFile.close()
outputFile.close()
```

**Now see comp.py**

# More ways to read a file

- Already saw: `for line in file` – to process each line as a separate string (inc. `'\n'` at ends)
- To get just a single line: `file.readline()`
  - Do it again to get the next line, and so on
- Also can get a list of lines as strings by `file.readlines()` – including `'\n'` at ends
- Or can even just `file.read()` – to get all of the file's text as a single string
- Note: `open` again if want to go back to the beginning of a file and read from start

**Try it!**

# Reading a file over the Internet

- Need a properly-formatted Uniform Resource Locator string – then open the remote file:

```
import urllib.request
urlName = "http://www.cs.ucsb.edu"
file = urllib.request.urlopen(urlName)
```

- Now treat it *almost* like any file open for reading:

```
for line in file:    # not okay – is not iterable
oneLine = file.readline()      # okay
allLines = file.readlines()    # okay
allText = file.read()          # okay
```

- If time: see getlines.py and outline.py
  - Include exception handling, and other stuff

# Stressing a point: `return x` vs. `print(x)`

- *Not the same thing!*
  - Python interpreter just makes it seem that way
- If a function returns a result, that result can be used later – for printing or whatever
  - e.g., `func1(value)` returns an integer, so:

```
>>> newResult = func1(5) + 92   # okay
```

- If a function prints a result – that's it: done
  - e.g., `func2()` prints, but doesn't return anything, so:

```
>>> result = func2()
... stuff gets printed here
>>> print(result)
None
```

# Repetition with a `while` loop

- `while` *condition*`:`

  `#` *executes over and over until false condition*
- Used for indefinite iteration – i.e., when no way to predict how many times it needs to execute
  - Use `for` loop for definite iteration (e.g., goes n times)
- Note 1: won't run at all if condition starts false
- Note 2: runs forever if condition stays true
- Sometimes helps to use `break` to exit loop, or `continue` to restart loop (work with `for` loops too)

# Applying `while`

**Try it!**
**(try `break` and `continue` too)**

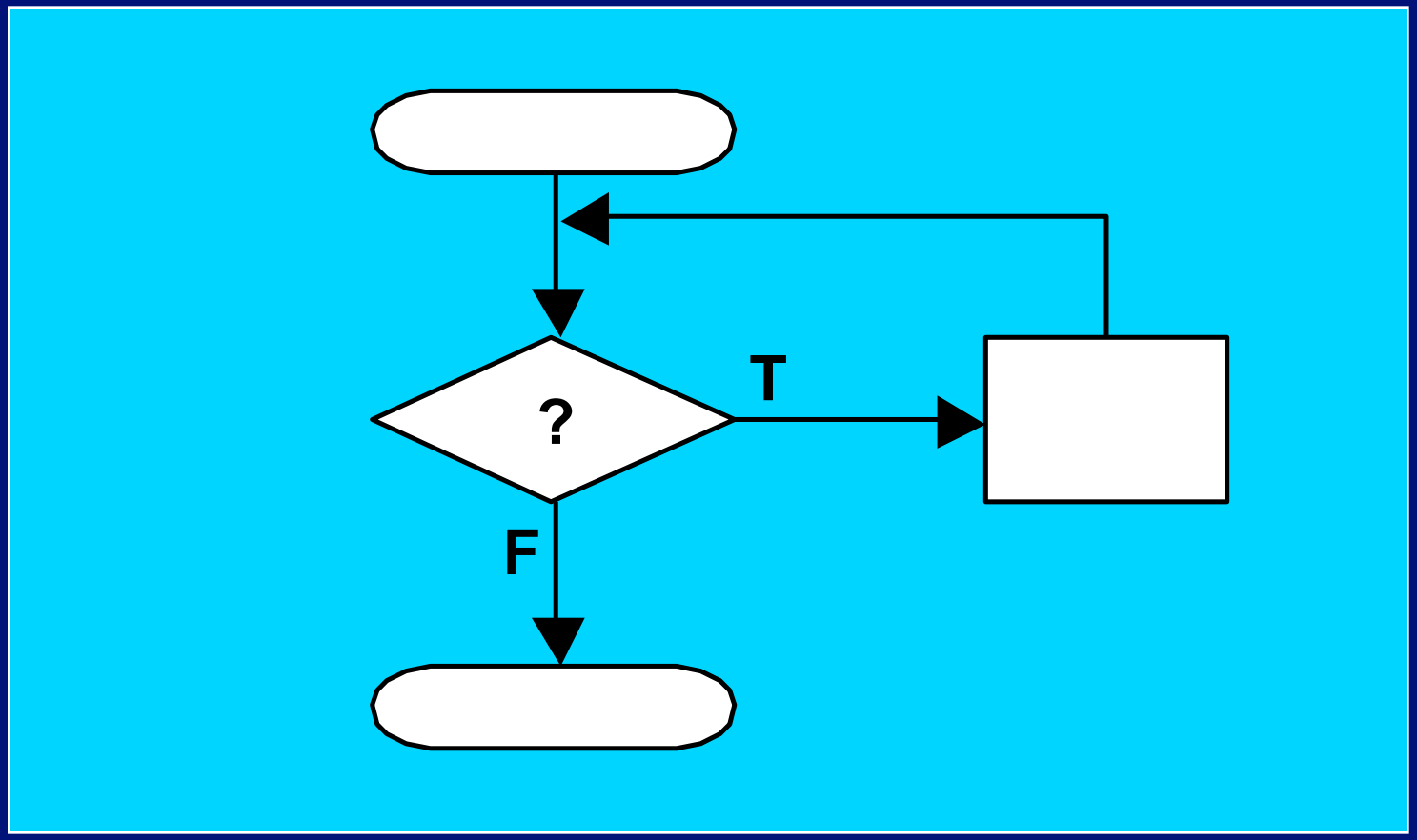
- Can be used for counter-controlled loops:

```
counter = 0                          # (1) initialize
while counter < n:                   # (2) check condition
    print(counter * counter)
    counter = counter + 1            # (3) change state
```

  - But this is a definite loop – easier to use `for`
- Better application – unlimited data entry:

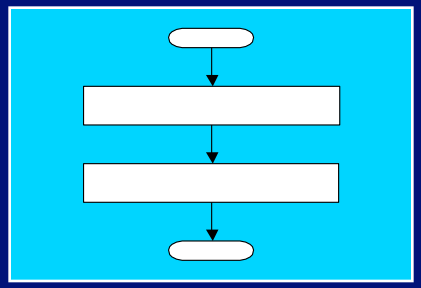
```
grade = input("enter grade: ")       # (1) initialize
while grade != "quit":               # (2) check condition
    # process grade here, then get next one
    grade = input("enter grade: ")   # (3) change state
```

# Flow of an iteration structure

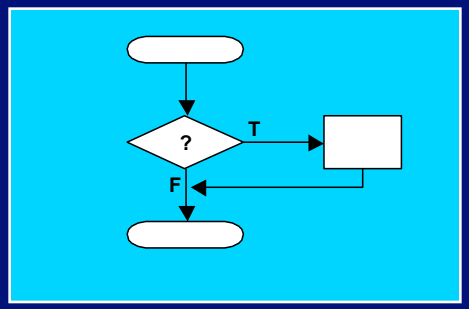


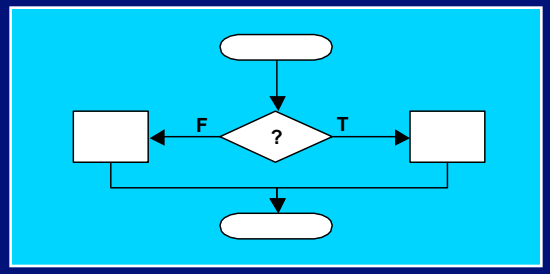
# Review: 3 control structure types

Sequence

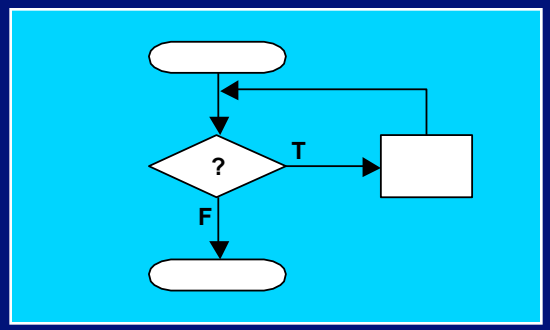


Selection




Iteration



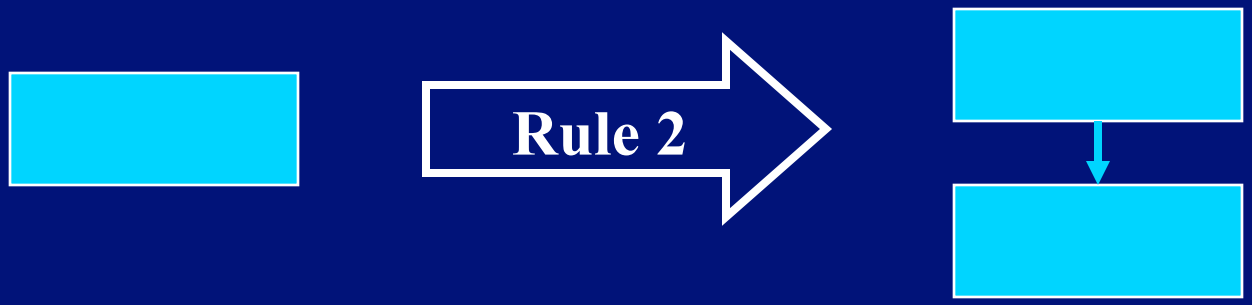
# Structure "rule" #1: start with the simplest flowchart


- Really just a way to start; clarifies the "big picture"
- For example:

  *get some data, calculate and then show some results*

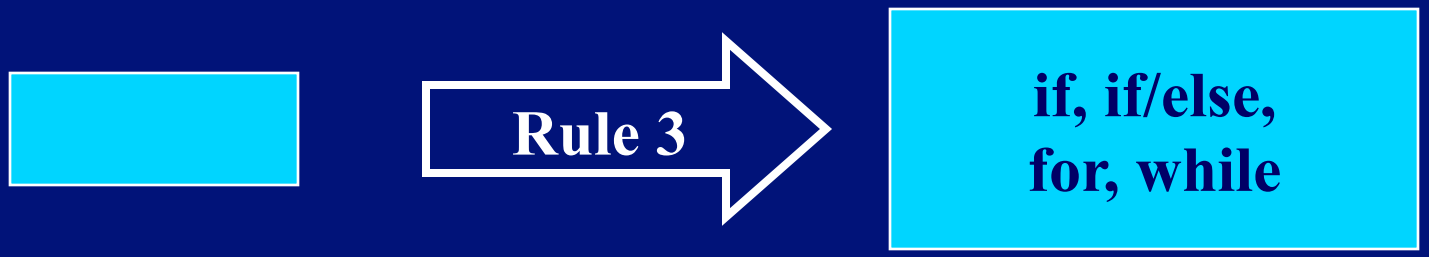
- Notice: just one rectangle

# Rule #2: replace any rectangle by two rectangles in sequence




- This "stacking rule" can apply repeatedly
- For example:
  1. Get data
  2. Process
  3. Show results

# Rule #3: replace any rectangle by any control structure




- This "nesting rule" also applies repeatedly – each control structure has its own rectangles
- e.g., nest a `while` loop in an `if` structure:

```
if  n > 0:
    while i < n:
        print(i)
        i = i + 1
```

# Rule #4: apply rules #2 and #3 repeatedly, and in any order

- Stack, nest, stack, nest, nest, stack, … gets more and more detailed as one proceeds
  - Think of control structures as building blocks that can be *combined in two ways only*.
- Overall process is known as "top-down design by stepwise refinement"
- Fact: *any algorithm* can be written as a combination of sequence, selection, and iteration structures.

# Formatted strings – old way

- *Overloaded* `%` operator – not just for modulus
  - Actually used two different ways:

```
>>> fString = "I have %d cents" % 42
>>> fString
'I have 42 cents'
```

  - First way is placeholder, second is format operator
- Placeholder actually is "conversion specifier"
  - Also `%f`, `%e`, `%g` for float; `%s` for string; and more – see Tables 5.2 and 5.3 (p. 162)
  - Can specify field width, left or right justify, other

**Try it!**

# New way: `str` method `format`

- Similar ideas, different syntax:
  `template.format(p0,p1,...,k0=v0,k1=v1,...)`
  - `template` is a string with conversion specifiers enclosed in curly braces; the `p`s are positional arguments and the `k=v` pairs are keyword arguments

```
>>> "{1} has ${0:.2f}".format(42,'Jo')
'Jo has $42.00'
```

  - All same conversion specifiers as old way
- Keyword arguments are handy, esp. if lots of args

```
>>> "{0} is {age}".format('Ed',age=20)
'Ed is 20'
```

*New way to format is not in text* – see http://www.python-course.eu/python3_formatted_output.php

# More about `print` (and `write`)

- Can use `print` function to write a file

```
>>> myfile = open("myfile.txt", "w")
>>> print("Hi!", file=myfile)
>>> myfile.close()
```

- By the way, can *append* text to a file too

```
>>> myfile = open("myfile.txt", "a")
>>> print("Hi again.", file=myfile)
```

- File method `write` does not format the output

```
>>> myfile.write("Another hi\n")  # must specify newline
11   # returns number of characters written (can only write just 1 string)
```

- Btw again: can change default format of `print` function

```
>>> print("No newline ", end="|/|")
No newline|/|>>> print("change", "separator", sep="-")
change-separator
```

# Next

Image processing