Programming Principles in Python (CSCI 503/490)

Control Statements

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Identifiers

- A sequence of letters, digits, or underscores, but…
- Also includes unicode "letters", spacing marks, and decimals (e.g. `Σ`)
- Must begin with a letter or underscore (_)
- Why not a number?
- Case sensitive (`a` is different from `A`)
- Conventions:
  - Identifiers beginning with an underscore (_) are reserved for system use
  - Use underscores (`a_long_variable`), **not** camel-case (`aLongVariable`)
  - Keep identifier names less than 80 characters
- Cannot be reserved words

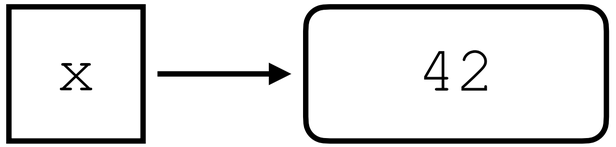
Types

- Don't worry about types, but think about types
- Variables can "change types"
  - `a = 0`
    `a = "abc"`
    `a = 3.14159`
- Actually, the name is being moved to a different value
- You can find out the type of the value stored at a variable `v` using `type(v)`
- Some literal types are determined by subtle differences
  - `1` vs `1.` (integer vs. float)
  - `1.43` vs `1.43j` (float vs. imaginary)
- Can do explicit type conversion (`int`, `str`, `float`)

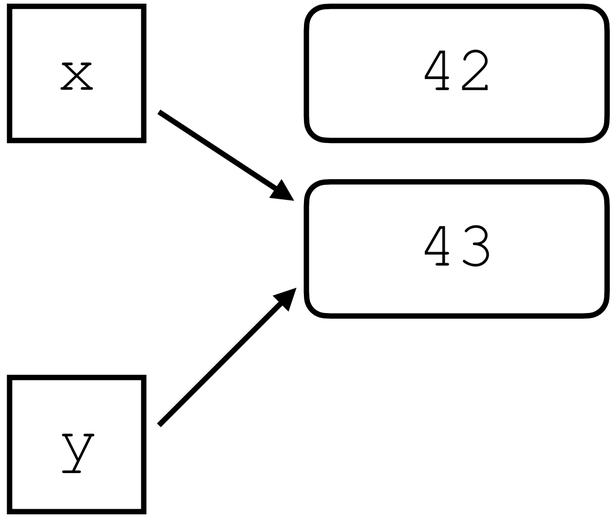
Assignment

- The = operator: `a = 34; c = (a + b) ** 2`
- Python variables are actually **pointers** to objects
- Also, augmented assignment: `+=`, `-=`, `*=`, `/=`, `//=`, `**=`

```
x = 42
```




```
x = x + 1
y = x
```



Simultaneous Assignment & Assignment Expressions

- Simultaneous assignment leaves less room for error:
  - `x,y = y,x`
- Assignment expressions use the walrus operator `:=`
  - `(my_pi := 3.14159) * r ** 2 + a ** 0.5/my_pi`
  - Use cases: if/while statement check than use, comprehensions

Assignment 1

- Due Tonight
- Get acquainted with Python using notebooks
- Make sure to follow instructions
  - Name the submitted file a1.ipynb
  - Put your name and z-id in the first cell
  - Label each part of the assignment using markdown
  - Make sure to produce output according to specifications
- ipynb files are in a JSON format. Please maintain the .ipynb extension!
- Questions?

Assignment 2

- Out soon

Quiz Wednesday

Control Statements

Boolean Expressions

- Type `bool`: `True` or `False`
- Note **capitalization**!
- Comparison Operators: <, <=, >, >=, ==, !=
  - Double equals (==) checks for equal values,
  - Assignment (=) assigns values to variables
- Boolean operators: `not`, `and`, `or`
  - Different from many other languages (`!`, `&&`, `||`)
- More:
  - `is`: exact same object (usually `a_variable is None`)
  - `in`: checks if a value is in a collection (`34 in my_list`)

if and else

- Blocks (suites) only executed if the condition is satisfied
- `if <boolean expression>:`
  `    <then-block>`
- `if <boolean expression>:`
  `    <then-block>`
  `else:`
  `    <else-block>`
- Remember **colon (:)**
- Remember **indentation**

-
```
if a < 34:
    b = 5
else:
    b = a - 34
```

elif is a shortcut

```
if a < 10:
    print("Small")
else:
    if a < 100:
        print("Medium")
    else:
        if a < 1000:
            print("Large")
        else:
            print("X-Large")
```

```
if a < 10:
    print("Small")
elif a < 100:
    print("Medium")
elif a < 1000:
    print("Large")
else:
    print("X-Large")
```

- Indentation is critical so else-if branches can become unwieldy (elif helps)

pass

- pass is a no-op
- Python doesn't allow an empty block so pass helps with this
- Used when commenting out code blocks
-
```
if a < 10:
    print("Small")
elif a < 100:
    print("Medium")
elif a < 1000:
    # print("Large") <- block would be empty (comments don't count)
    pass
else:
    print("X-Large")
```

while

- while repeats the execution of the block
- `while <boolean expression>:`
  `    <loop-block>`
- Condition is checked at the beginning and before each repeat
- If condition is `False`, loop will never execute
- Don't use a while loop to iterate (use for loop instead)
- Example:

```
- d = 100
  while d > 0:
      a = get_next_input()
      d -= a
```

break and continue

- `break`: immediately exit the current loop
- `continue`: stop loop execution and go back to the top of the loop, checking the condition again
-
```
while d > 0:
    a = get_next_input()
    if a > 100:
        break
    if a < 10:
        continue
    d -= a
```
- These are similar to goto statements in that they can jump from one statement to another part of the code but scoped to the current loop

The Go To Statement Debate

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of

"…I became convinced that the go to statement should be abolished from all 'higher level' programming languages… The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."

been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather

namic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

[Dijkstra, 1968]

The Go To Statement Debate

```
for i := 1 to n
do begin
  for j := 1 to n do
    if x[i, j]<>0
      then goto reject;
  writeln
('The first all-zero
                  row is ', i);
  break;
reject: end;
```

```
i := 1;
repeat
  j := 1;
  allzero := true;
  while (j<=n) and allzero
  do begin
    if x[i, j]<>0
      then allzero := false;
    j := j + 1;
  end;
  i := i + 1;
until (i>n) or allzero;
if i<=n
    then writeln
('The first all-zero
                row is ', i - 1);
```

```
i := 1;
repeat
  j := 1;
  while (j<=n)
  and (x[i, j] = 0) do
    j := j + 1;
  i := i + 1;
until (i>n) or (j>n);
if j>n
    then writeln
('The first all-zero
                row is', i - 1);
```

"All of my experiences compel me to conclude that it is time to part from the dogma of GOTO-less programming. It has failed to prove its merit"

[Rubin, 1987]

Programming Principles: break, continue, goto

- ACM the published a number of critiques of Rubin's letter, Dijkstra also wrote some notes on this: bugs, maybe the language is bad…
- Most computer scientists agree that the problem was over-use, not that the statement is never useful
- Break and continue are more structured gotos because they apply only to the current block
- Breaks and continues at the top of a loop are better
- Multi-level breaks are annoying (compare with return statements in functions)

Continue at the beginning of a loop

- Like elif, can help with indentation
-
```
while x >= 0:
    d = get_data()
    if d is not None:
        # do stuff
```

-
```
while x >= 0:
    d = get_data()
    if d is None:
        continue
    # do stuff
```

Loop Styles

- Loop-and-a-Half

```
d = get_data() # priming rd
while check(d):
    # do stuff
    d = get_data()
```

- Infinite-Loop-Break

```
while True:
    d = get_data()
    if not check(d):
        break
    # do stuff
```

- Better way?

Loop Styles

- Loop-and-a-Half

```
d = get_data() # priming rd
while check(d):
    # do stuff
    d = get_data()
```

- Infinite-Loop-Break

```
while True:
    d = get_data()
    if check(d):
        break
    # do stuff
```

- Assignment Expression (Walrus)

```
while check(d := get_data()):
    # do stuff
```

do-while

- do-while loops always execute at least once
- There is no do-while loop construct in Python
- Can set the condition so that it is always True first time through the loop
- …or move the break to the end of the loop

Looping Errors

```
# while loop - summing the numbers 1 to 10
n = 10
cur_sum = 0
# sum of n numbers
i = 0
while  i <= n:
    i = i + 1
    cur_sum = cur_sum + i

print("The sum of the numbers from 1 to", n, "is ", cur_sum)
```

Looping Errors

```
# while loop - summing the numbers 1 to 10
n = 10
cur_sum = 0
# sum of n numbers
i = 0
while  i <= n:
    cur_sum = cur_sum + i
    i = i + 1

print("The sum of the numbers from 1 to", n, "is ", cur_sum)
```

Looping Errors

```
# while loop - summing the numbers 1 to 10
n = 10
cur_sum = 0
# sum of n numbers
i = 0
while  i != n:
    cur_sum = cur_sum + i
    i = i + 1

print("The sum of the numbers from 1 to", n, "is ", cur_sum)
```

For Loop

- for loops in Python are really **for-each** loops
- Always an element that is the current element
  - Can be used to iterate through iterables (containers, generators, strings)
  - Can be used for counting
- ```
  for i in range(5):
      print(i) # 0 1 2 3 4
  ```
- `range(5)` **generates** the numbers 0,1,2,3,4
  ```

Range

- Python has lists which allow enumeration of all possibilities: [0,1,2,3,4]
- Can use these in for loops
- ```
  for i in [0,1,2,3,4]:
      print(i) # 0 1 2 3 4
  ```
- **but** this is less efficient than range (which is a generator)
- ```
  for i in range(5):
      print(i) # 0 1 2 3 4
  ```
- List must be stored, range doesn't require storage
- Printing a range doesn't work as expected:
  - `print(range(5)) # prints "range(0, 5)"`
  - `print(list(range(5)) # prints "[0, 1, 2, 3, 4]"`

Range

- Different method signatures
  - `range(n)` → `0, 1, …, n-1`
  - `range(start, end)` → `start, start + 1, …, end - 1`
  - `range(start, end, step)`
    → `start, start + step, … < end`
- Negative steps:
  - `range(0,4,-1) # <nothing>`
  - `range(4,0,-1) # 4 3 2 1`
- Floating-point arguments are **not** allowed

Looping Errors

```
# for loop - summing the numbers 1 to 10
n = 10
cur_sum = 0
for i in range(n):
    cur_sum += i

print("The sum of the numbers from 1 to", n, "is ", cur_sum)
```

Looping Errors

```
# for loop - summing the numbers 1 to 10
n = 10
cur_sum = 0
for i in range(n+1):
    cur_sum += i

print("The sum of the numbers from 1 to", n, "is ", cur_sum)
```

Looping Errors

```
# for loop - summing the numbers 1 to 10
n = 10
cur_sum = 0
for i in range(1, n+1):
    cur_sum += i

print("The sum of the numbers from 1 to", n, "is ", cur_sum)
```