

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

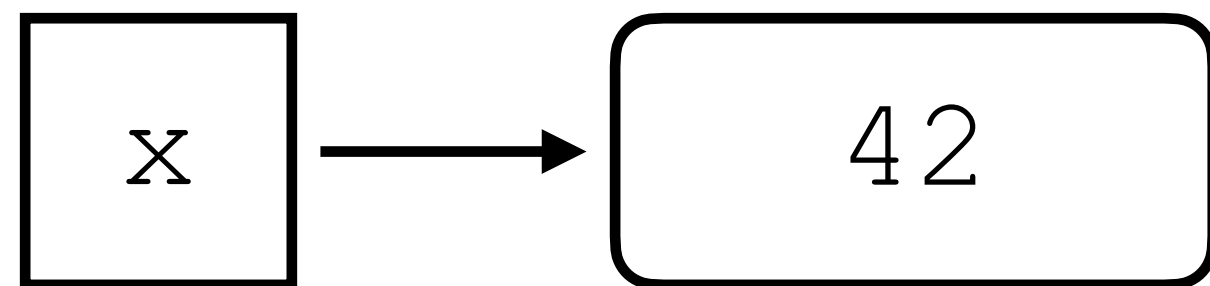
Types

- 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`)

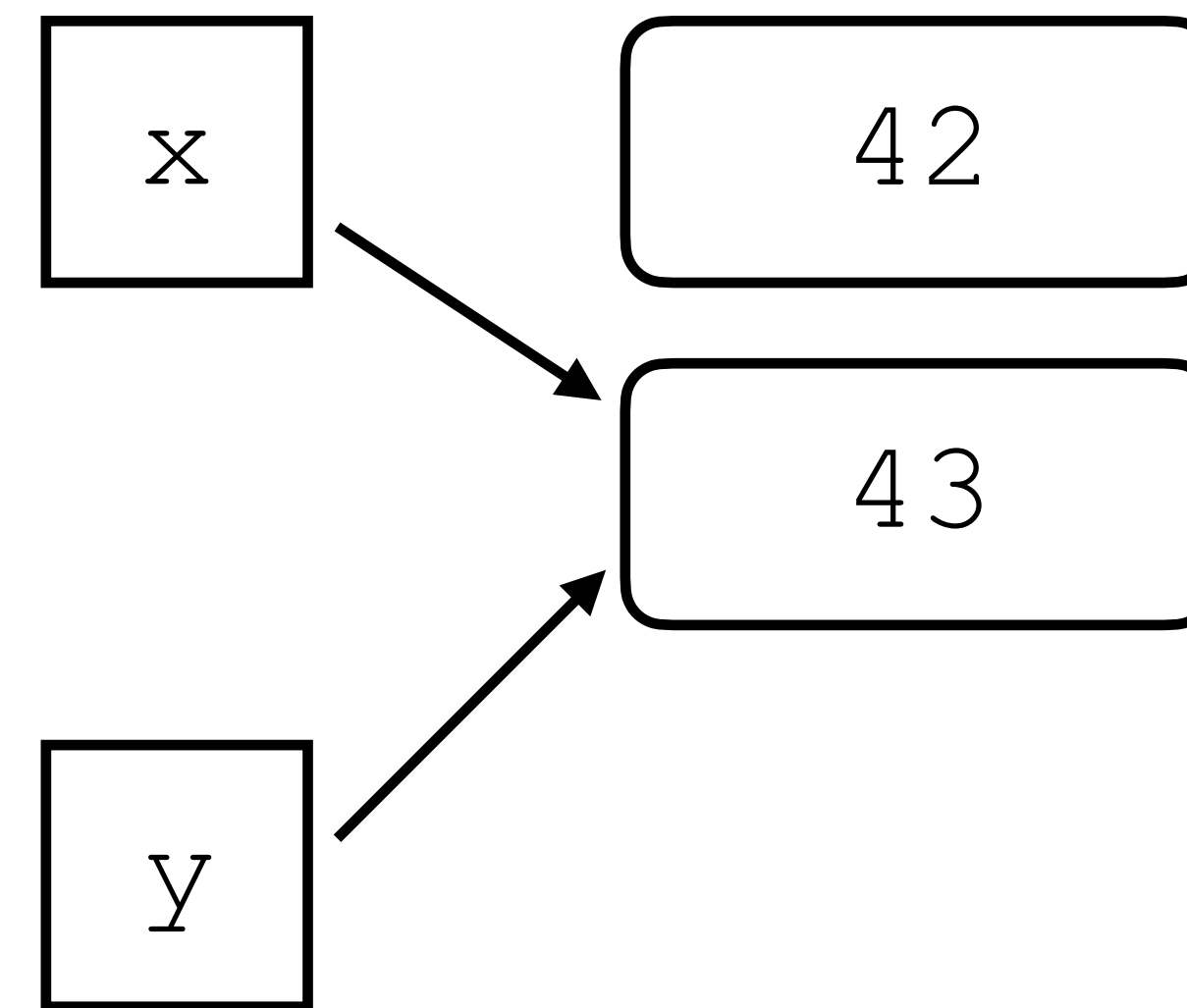
Assignment

- Python variables are actually **pointers** to objects (names for values)

```
x = 42
```



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



Simultaneous Assignment

- In most languages, this requires another variable
 - `x_old = x`
`x = y`
`y = x_old`
- Simultaneous assignment leaves less room for error:
 - `x, y = y, x`
- Also useful for unpacking a collection of values:
 - `dateStr = "03/08/2014"`
`monthStr, dayStr, yearStr = dateStr.split("/")`

Assignment Expressions

- AKA the "walrus" operator `:=`
- Names a value that can be used but also referenced in the rest of the expression
- `(my_pi := 3.14159) * r ** 2 + a ** 0.5/my_pi`
- Use cases: if/while statement check then use, comprehensions
- Supported in Python 3.8+

Assignment 2

- Collatz Conjecture Variant: $7n \pm 1$
- Control Flow and Functions
- Do not use sequences, other collections, or comprehensions for this assignment

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

dynamic 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 then 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 d >= 0:
    d = get_data()
    if d is not None:
        # do stuff
```

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

# Read and Check Data Loop

---

- Suppose I have two functions:
  - `get_data()` : returns the next data item `d`
  - `check(d)` : checks whether the data item is valid
- Want to build a loop that reads and prints data items until `check(d)` fails

# 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 not 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
- So...
  - can set the condition so that it is always True first time through the loop
  - ...or create an if-break at the end of the loop (infinite-loop-break)

# 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)`  $\rightarrow 0, 1, \dots, n-1$
  - `range(start, end)`  $\rightarrow start, start + 1, \dots, end - 1$
  - `range(start, end, step)`  
 $\rightarrow start, start + step, \dots < end$
- 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)
```

Functions

Functions

- Call a function `f`: `f (3)` or `f (3, 4)` or ... depending on number of parameters
- `def <function-name> (<parameter-names>):`
 `"""Optional docstring documenting the function"""`
 `<function-body>`
- `def` stands for function definition
- docstring is convention used for documentation
- Remember the **colon** and **indentation**
- Parameter list can be empty: `def f(): ...`

Functions

- Use `return` to return a value
- `def <function-name> (<parameter-names>) :`
 `# do stuff`
 `return res`
- Can return more than one value using commas
- `def <function-name> (<parameter-names>) :`
 `# do stuff`
 `return res1, res2`
- Use **simultaneous assignment** when calling:
 - `a, b = do_something(1, 2, 5)`
- If there is no return value, the function returns `None` (a special value)

Default Values & Keyword Arguments

- Can add `=<value>` to parameters
- ```
def rectangle_area(width=30, height=20):
 return width * height
```
- All of these work:
  - `rectangle_area()` # 600
  - `rectangle_area(10)` # 200
  - `rectangle_area(10, 50)` # 500
- If the user does not pass an argument for that parameter, the parameter is set to the default value
- Can also pass parameters using `<name>=<value>` (keyword arguments):
  - `rectangle_area(height=50)` # 1500

# Return

---

- As many return statements as you want
- Always end the function and go back to the calling code
- Returns do not need to match one type/structure (generally not a good idea)
- ```
def f(a,b):  
    if a < 0:  
        return -1  
    while b > 10:  
        b -= a  
        if b < 0:  
            return "BAD"  
    return b
```