Programming Principles in Python (CSCI 503)

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. \( \Sigma \))
• 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 :=
  - \((my\_pi := 3.14159) \times r \times 2 + a \times 0.5/my\_pi\)
  - Use cases: if/while statement check than use, comprehensions
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`)
Assignment 2

• Due next Monday
• Python control flow and functions
• Do not use containers like lists!
• The 7x+-1 function
• Make sure to follow instructions
  - Name the submitted file a2.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
if and else

- Blocks 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

```python
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
• \( \text{while } \langle \text{boolean expression} \rangle: \)
  \(<\text{loop-block}>\)
• Condition is checked at the beginning and before each repeat
• If condition is \texttt{False}, loop will never execute
• Don't use a while loop to iterate (use for loop instead)
• Example:
  
  - \( \text{d} = 100 \)
  - \( \text{while d > 0:} \)
    - \( \text{a} = \text{get\_next\_input()} \)
    - \( \text{d} \text{ -= a} \)
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
"...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."

[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);
```

"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

```python
while x >= 0:
    d = get_data()
    if d is not None:
        # do stuff
```

```python
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 check(d):
      break
    # do stuff

• Better way?
Loop Styles

• Loop-and-a-Half

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

• Infinite-Loop-Break

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

• Assignment Expression (Walrus)

```python
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
  ```python
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
  
  ```python
  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 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
  ```python
  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)
  ```python
  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:
  ```python
  - 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, n) → start, start + 1, ..., start + (n-1)
  - range(start, n, step)
    → start, start + step, ..., start + (n-1)*step

• 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)
Loops

- # 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
  
  ```python
  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

- Call a function \( f: f(3) \) or \( f(3, 4) \) or ... depending on number of parameters
- \texttt{def} \texttt{<function-name>\((<parameter-names>)\):}
  
  """Optional docstring documenting the function"

- \texttt{def} stands for function definition
- docstring is convention used for documentation
- Remember the \texttt{colon} and \texttt{indentation}
- Parameter list can be empty: \texttt{def f(): ...}
Functions

• Use `return` to return a value

```python
def <function-name>(<parameter-names>):
    # do stuff
    return res
```

• Can return more than one value using commas

```python
def <function-name>(<parameter-names>):
    # do stuff
    return res1, res2
```

• Use **simultaneous assignment** when calling:
  ```python
  - a, b = do_something(1,2,5)
  ```

• If there is no return value, the function returns `None` (a special value)