Programming Principles in Python (CSCI 503/490)

Sequences

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)
if, else, elif, pass

• if \( a < 10 \):
  
  print("Small")

else:
  
  if \( a < 100 \):
    
    print("Medium")

else:
  
    if \( a < 1000 \):
      
      print("Large")

else:
  
    print("X-Large")

• Indentation is critical so else-if branches can become unwieldy (elif helps)
• Remember colons and indentation
• \texttt{pass} can be used for an empty block
while, break, continue

• while <boolean expression>:  
  <loop-block>

• Condition is checked at the **beginning** and before each repeat

• **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
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

"...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."

---

Edgar Dijkstra: Go To Statement Considered Harmful

The Go To Statement Debate

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 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.

[Dijkstra, 1968]
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
  ```
TA Change

- New TA: Eswar Gottuparthi
- Office Hours will Change
- Web page to be Updated
- Similar Policies
Assignment 2

- Due next Wednesday
- Python control flow and functions
- Do not use containers like lists!
- Check Collatz Conjecture
- 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
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
  ```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:
  - `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)
# 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

• 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
  ```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)
Default Values & Keyword Arguments

• Can add \texttt{=\textless value\textgreater} to parameters
  
  
  \textbf{def rectangle\_area(width=30, height=20):}
  \hspace{1em} return width * height

• All of these work:
  
  - \texttt{rectangle\_area()} \# 600
  - \texttt{rectangle\_area(10)} \# 200
  - \texttt{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 \texttt{<name>=<value>} (keyword arguments):
  
  - \texttt{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)

```python
def f(a, b):
    if a < 0:
        return -1
    while b > 10:
        b -= a
    if b < 0:
        return "BAD"
    return b
```
Sequences

• Strings are sequences of characters: "abcde"
• Lists are also sequences: [1, 2, 3, 4, 5]
• Tuples: (1, 2, 3, 4, 5)
Lists

• Defining a list:  
  `my_list = [0, 1, 2, 3, 4]`

• But lists can store different types:
  - `my_list = [0, "a", 1.34]`

• Including other lists:
  - `my_list = [0, "a", 1.34, [1, 2, 3]]`
• Defining a tuple: \texttt{my\_tuple = (0, 1, 2, 3, 4)}

• But tuples can store different types:
  - \texttt{my\_tuple = (0, "a", 1.34)}

• Including other tuples:
  - \texttt{my\_tuple = (0, "a", 1.34, (1, 2, 3))}

• How do you define a tuple with \texttt{one} element?
Lists Tuples

• Defining a tuple: `my_tuple = (0, 1, 2, 3, 4)`
• But tuples can store different types:
  - `my_tuple = (0, "a", 1.34)`
• Including other tuples:
  - `my_tuple = (0, "a", 1.34, (1, 2, 3))`
• How do you define a tuple with one element?
  - `my_tuple = (1)`  # doesn't work
  - `my_tuple = (1,)`  # add trailing comma
List Operations

- **Not** like vectors or matrices!
- Concatenate: \([1, 2] + [3, 4] \# [1,2,3,4]\)
- Repeat: \([1,2] \times 3 \# [1,2,1,2,1,2]\)
- Length: `my_list = [1,2]; len(my_list) \# 2`
List Sequence Operations

• Concatenate: \([1, 2] + [3, 4] \# [1,2,3,4]\)

• Repeat: \([1,2] \times 3 \# [1,2,1,2,1,2]\)

• Length: \(\text{my\_list} = [1,2]; \ \text{len(my\_list)} \# 2\)

• Concatenate: \((1, 2) + (3, 4) \# (1,2,3,4)\)

• Repeat: \((1,2) \times 3 \# (1,2,1,2,1,2)\)

• Length: \(\text{my\_tuple} = (1,2); \ \text{len(my\_tuple)} \# 2\)

• Concatenate: \"ab\" + \"cd\" \# \"abcd\"

• Repeat: \"ab\" \times 3 \# \"ababab\"

• Length: \(\text{my\_str} = \"ab\"; \ \text{len(my\_str)} \# 2\)
Sequence Indexing

• Square brackets are used to pull out an element of a sequence
• We always start counting at zero!

my_str = "abcde"; my_str[0] # "a"
my_list = [1,2,3,4,5]; my_list[2] # 3
my_tuple = (1,2,3,4,5); my_tuple[5] # IndexError
Negative Indexing

- Subtract from the end of the sequence to the beginning
- We always start counting at zero -1 (zero would be ambiguous!)

- `my_str = "abcde"; my_str[-1] # "e"
- `my_list = [1,2,3,4,5]; my_list[-3] # 3`
- `my_tuple = (1,2,3,4,5); my_tuple[-5] # 1`
Slicing

- Want a subsequence of the given sequence
- Specify the start and the first index not included
- Returns the same type of sequence

- `my_str = "abcde"; my_str[1:3] # "bc"
- `my_list = [1,2,3,4,5]; my_list[3:4] # [4]
- `my_tuple = (1,2,3,4,5); my_tuple[2:99] # (3,4,5)
Negative Indices with Slices

• Negative indices can be used instead or with non-negative indices
  
  • `my_str = "abcde"; my_str[-4:-2] # "bc"
  
  • `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]
  
  • How do we include the last element?
  
  • `my_tuple = (1,2,3,4,5); my_tuple[-2:]`
Negative Indices with Slices

- Negative indices can be used instead or with non-negative indices
  - `my_str = "abcde"; my_str[-4:-2] # "bc"
  - `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]
- How do we include the last element?
  - `my_tuple = (1,2,3,4,5); my_tuple[-2:]`
Implicit Indices

• Don't need to write indices for the beginning or end of a sequence
• Omitting the first number of a slice means start from the beginning
• Omitting the last number of a slice means go through the end

```python
my_tuple = (1,2,3,4,5); my_tuple[-2:len(my_tuple)]  # (4, 5)
my_tuple = (1,2,3,4,5); my_tuple[-2:]  # (4, 5)
```

• Can create a **copy** of a sequence by omitting both

```python
my_list = [1,2,3,4,5]; my_list[:])  # [1, 2, 3, 4, 5]
```
Iteration

- for d in sequence:
  # do stuff

**Important:** d is a data item, not an index!

- sequence = "abcdef"

  for d in sequence:
  
  print(d, end=" ")              # a b c d e f

- sequence = [1,2,3,4,5]

  for d in sequence:
  
  print(d, end=" ")              # 1 2 3 4 5

- sequence = (1,2,3,4,5)

  for d in sequence:
  
  print(d, end=" ")              # 1 2 3 4 5