Programming Principles in Python (CSCI 503/490)

Sequences

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Quiz

Quiz

1. What type of statement did Dijkstra “consider harmful”?
   (a) do-while
   (b) break
   (c) goto
   (d) continue

Quiz

2. What does `4 + 9 / 2 - 2` evaluate to?
   (a) `4`
   (b) `6`
   (c) `6.5`
   (d) `4.5`

Quiz

3. Which expression computes whether `a` is greater than `2` and `b` is not equal to `10000`?

   (a) `a > 2 && b != 10000`

   (b) `a > 2 and b is not 10000`

   (c) `not (a <= 2 or b == 10000)`

   (d) `!(a <= 2 || b == 10000)`

Quiz

4. Which is an **invalid** string?
   a. `'''It's a dog's life'''`
   b. `'It\'s a dog\'s life'`
   c. `"It's a dog's life"`
   d. `"""It's a dog's life'''`

Quiz

5. Which is **not** a **valid** python identifier?
   (a) `mañana`
   (b) `_in_order_list`
   (c) `int`
   (d) `3bears`

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")
```

```
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)
- Remember colons and indentation
- `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

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]

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
```

Assignment 2

- Due next Wednesday
- Python control flow and functions

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)
```

[The Carpentries, CC-BY 4.0]

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)
```

[The Carpentries, CC-BY 4.0]

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)
```

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
```

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
```

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]]`

~~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?

~~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] * 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] * 3 # [1,2,1,2,1,2]`
- Length: `my_list = [1,2]; len(my_list) # 2`

- Concatenate: `(1, 2) + (3, 4) # (1,2,3,4)`
- Repeat: `(1,2) * 3 # (1,2,1,2,1,2)`
- Length: `my_tuple = (1,2); len(my_tuple) # 2`

- Concatenate: `"ab" + "cd" # "abcd"`
- Repeat: `"ab" * 3 # "ababab"`
- Length: `my_str = "ab"; 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`