Conditional branching, looping, and flow

Allesina S, Wilmes M (2019). *Computing Skills for Biologists.* Princeton UP 3.5 – 3.6

Week 09 | PP8300 | 2021

Zachary Konkel

Some more basic functions

`max()` – returns max value of a string, list, or tuple

`min()`

`sum()` – returns sum of elements of list or set

`in`

`in` – Tests for membership and returns `True` or `False`

```
In [7]: "s" in "string"
Out[7]: True
```

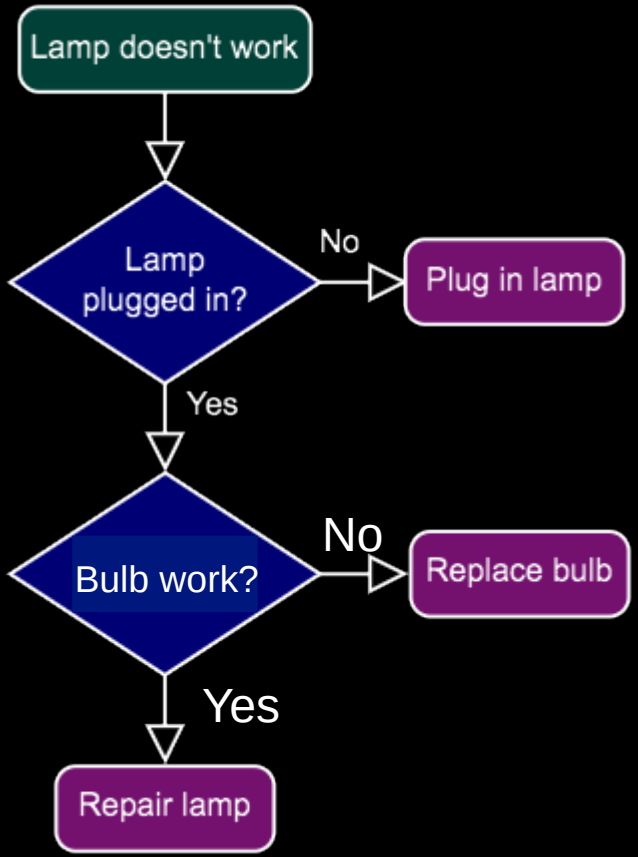
`in`

`in` – Tests for membership and returns `True` or `False`

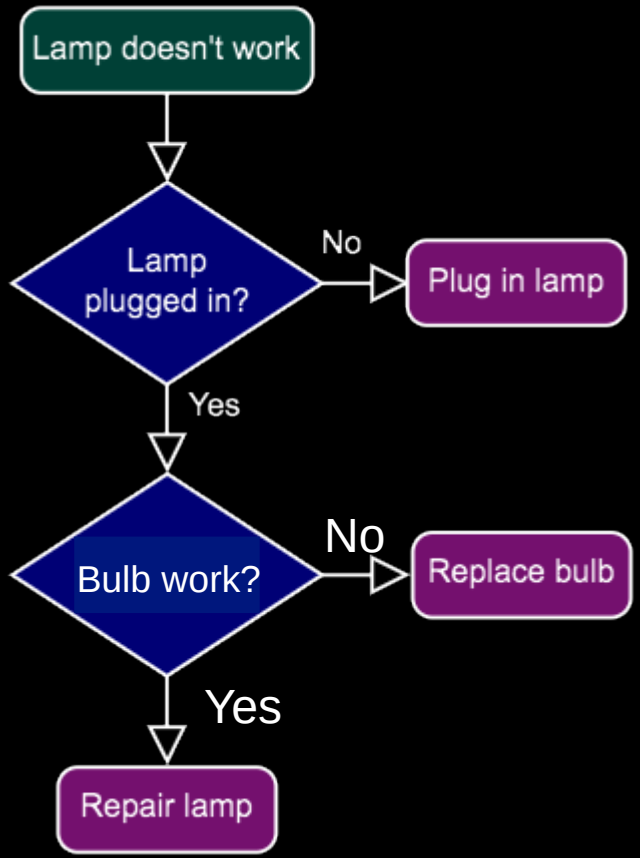
`not` – Applies the opposite test

```
In [7]: "s" in "string"
Out[7]: True
In [8]: 36 not in [1, 2, 36]
Out[8]: False
In [9]: (1, 2) in [(1, 3), (1, 2), 1000, 'aaa']
Out[9]: True
# for dictionaries, you can test whether a key exists
In [10]: "z" in {"a": 1, "b": 2, "c": 3}
Out[10]: False
In [11]: "c" in {"a": 1, "b": 2, "c": 3}
Out[11]: True
```

Conditional Branching

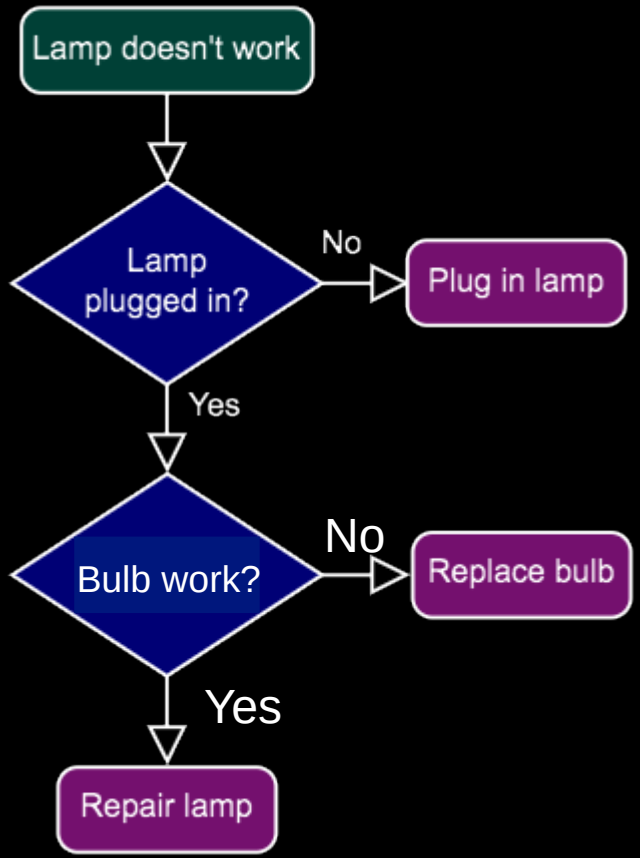


Conditional Branching



Let's say we have an object `lamp` that we try to turn on with a method… if it is on `lamp` evaluates to `True`, if it is off then it evaluates to `False`

Conditional Branching



Let's say we have an object `lamp` that we try to turn on with a method… if it is on `lamp` evaluates to `True`, if it is off then it evaluates to `False`

```
lamp.turn_on_lamp()
if not lamp:
    print("Lamp doesn't work")
    if lamp.plug_in():
        print("Lamp is plugged in")
        if lamp.bulb():
            print("Repairing lamp")
            lamp.repair()
        else:
            print("Replacing bulb")
            lamp.replace_bulb()
    else:
        print("Plugging in lamp")
        lamp.plug_in()
```

Flow & conditional branching in python

```
x = 4

if x % 2 == 0:
    print("Divisible by 2")
```

Flow & conditional branching in python

```
x = 4

if x % 2 == 0:
    print("Divisible by 2")
```

Note the ‘:’

- Commands that alter *the flow* of a program contain ‘:’

  `if, elif, else, for, while`

Flow & conditional branching in python

```
x = 4

if x % 2 == 0:
    print("Divisible by 2")
```

Note the **':'**

- Commands that alter *the flow* of a program contain ':'

  `if, elif, else, for, while`

Note the **indentation** on the following line

- Indentation tells python to execute the line(s) *if the conditional expression evaluates* `True`
- Python will accept most consistent indentation, but recommends four spaces
  - *DO NOT USE TABS* unless your text editor converts Tab to four spaces
- All code at the same indentation level is considered part of the same block of code

Multiple conditions in conditional branching

```
if x % 2 == 0:
    print("Divisible by 2")
else:
    print("Not divisible by 2")
```

`else:` if all of the above conditions are not met, execute the `else` block

- It is NOT required

Multiple conditions in conditional branching

```
x = 17

if x % 2 == 0:
    print("Divisible by 2")
elif x % 3 == 0:
    print("Divisible by 3")
elif x % 5 == 0:
    print("Divisible by 5")
elif x % 7 == 0:
    print("Divisible by 7")
else:
    print("Not divisible by 2, 3, 5, 7")
```

`elif:` if the preceding condition is not `True` AND the `elif` condition is, then run the `elif` block

- Otherwise, proceed to the next condition if any is present

Looping in python

Looping modifies the flow of a program by iterating through a block of code multiple times

- Iterating through files in a directory, elements in a list, etc

Looping in python

Python employs two loop types: `for, while`

- `for` will loop through elements of a sequence in order
  - Your program knows beforehand how long you want to loop (e.g. through a list, keys in a dictionary)

```
# print the characters of a string
my_string = "a given string"
for character in my_string:
    print(character)
```

Looping in python

Python employs two loop types: `for, while`

- `while` will loop so long as a condition is met
  - You don't directly know how long you want to loop
  - This can be infinite! `CTRL + C` to stop (`CMD + C`)

```
# print the first few Fibonacci numbers
a = 1
b = 1
c = 0
while c < 10000:
    c = a + b
    a = b
    b = c
    print(c)
```

Altering loop under specific conditions

Let's say you don't want to continue looping under a specific condition

- python has 3 mechanisms to address this:
  - `continue, break, pass`
    - `continue` – continue to next iteration in loop
    - `break` – stop and break out of the loop
    - `pass` – a null placeholder, rarely used

Altering loop under specific conditions

Let's say you don't want to continue looping under a specific condition

- python has 3 primary ways to address this:
  - `continue, break, pass`
    - `continue` – continue to next iteration in loop

```
# list the first 100 even numbers
x = 0
found = 0
while found < 100:
    x = x + 1
    if x % 2 == 1:
        continue
    print(x)
    found = found + 1
```

Altering loop under specific conditions

Let's say you don't want to continue looping under a specific condition

- python has 3 primary ways to address this:
  - `continue, break, pass`
    - `break` – stop and break out of the loop

```
# find the first integer >= 15000 that is divisible by 19
x = 15000
while x < 19000:
    if x % 19 == 0:
        print(str(x)  + " is divisible by 19")
        break
    x = x + 1
```

Putting it together – practical looping

python presents multiple ways to iterate through a sequence

- remember there should only be one and preferably only one obvious solution

```
# print x^2 for x in 0 to 9
for x in range(10):
    print(x ** 2)
```

Putting it together – practical looping

python presents multiple ways to iterate through a sequence

- remember there should only be one and preferably only one obvious solution

```
# print x^2 for x in 0 to 9
for x in range(10):
    print(x ** 2)
```

```
for k, x in enumerate(my_string):
    print(k, x)
```

```
z = [1, 5, "mystring", True]
for element, value in enumerate(z):
    print("element: " + str(element) + " value: " +
        ↳ str(value))
```

Intermezzo 3.3

Understanding loops is fundamental to writing efficient programs. Take a moment to consolidate what you have learned and determine how many times "hello" will be printed in the following small programs. Try to come up with the answer before running the code in Python:

(a)
```
for i in range(3, 17):
    print("hello")
```

(b)
```
for j in range(12):
    if j % 3 == 0:
        print("hello")
```

(c)
```
for j in range(15):
     if j % 5 == 3:
        print("hello")
     elif j % 4 == 3:
        print("hello")
```

(d)
```
z = 0
while z != 15:
    print("hello")
    z = z + 3
```

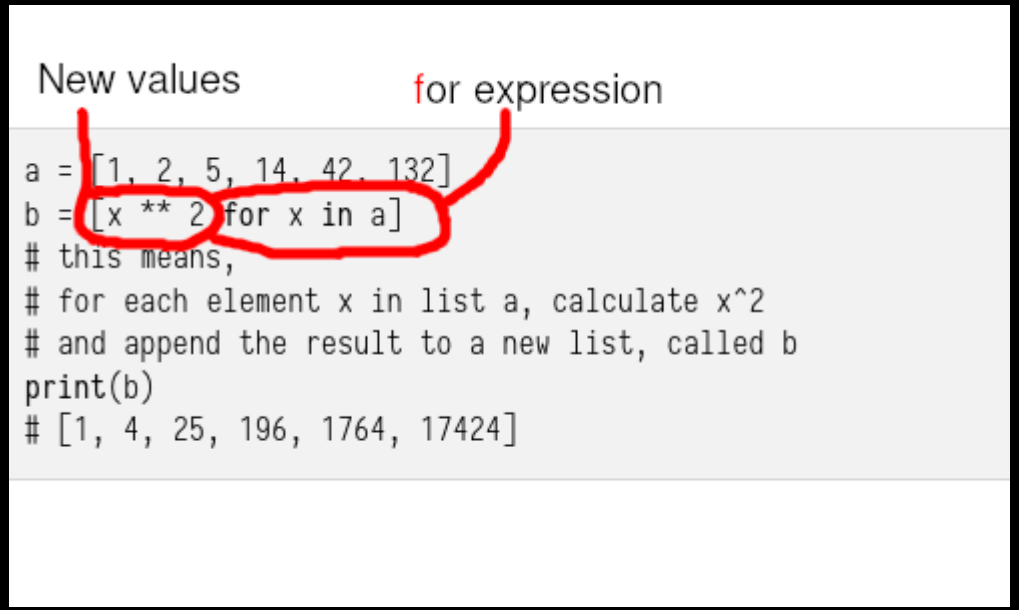
List comprehensions abridge loop functions

python provides a convient, readable way to consolidate list expressions into a single line

```
a = [1, 2, 5, 14, 42, 132]
b = [x ** 2 for x in a]
# this means,
# for each element x in list a, calculate x^2
# and append the result to a new list, called b
print(b)
# [1, 4, 25, 196, 1764, 17424]
```

List comprehensions abridge loop functions

python provides a convient, readable way to consolidate list expressions into a single line of code



List comprehensions abridge loop functions

python provides a convient, readable way to consolidate list expressions into a single line of code

- List comprehensions can include simple conditional expressions too

```
x = [5, 6, 600, 500]

w = [i for i in x if i < 100]
```

List comprehensions abridge loop functions

python provides a convient, readable way to consolidate list expressions into a single line of code

- List comprehensions can include simple conditional expressions too

```
x = [5, 6, 600, 500]

w = [i for i in x if i < 100]
```

```
x, w = [5, 6, 600, 500], []
for i in x:
    if i < 100:
        w.append(i)
```

List comprehensions abridge loop functions

List comprehensions can also be used to construct dictionaries and other data types

```
x = [‘honey’, ‘meat’, ‘juice’]
y = [‘bee’, ‘bear’, ‘berry’]
z = {element: y[index] for index,element in enumerate(x)}
 {'honey': 'bee', 'meat': 'bear', 'juice': 'berry'}
```

List comprehensions abridge loop functions

List comprehensions can also be used to construct dictionaries and other data types

```
x = [‘honey’, ‘meat’, ‘juice’]
y = [‘bee’, ‘bear’, ‘berry’]
z = {element: y[index] for index,element in enumerate(x)}
 {'honey': 'bee', 'meat': 'bear', 'juice': 'berry'}
```

My opinion: list comprehensions are useful in constructing neat, readable python code; however, they simply implement and draw from conceptually understanding loops and conditional expressions, so I think it is best to familiarize yourself with the longer format first.

What can looping and conditional branching enable in your research?