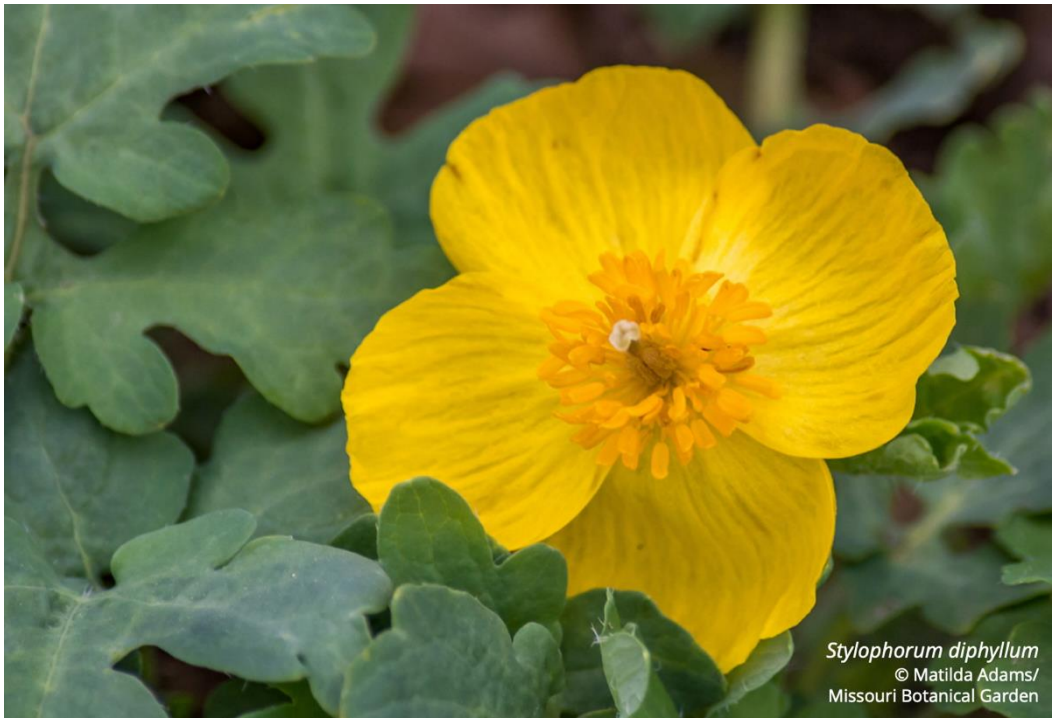
Lecture 14
Python: containers, loops, script arguments



Course: Practical Bioinformatics (BIOL 4220)
Instructor: Michael Landis
Email: michael.landis@wustl.edu

Lecture 14 outline

Last time: variables, operators, containers, functions

This time: Python (2 of 3)

Python

- if-statements
- for-loops
- more with containers

if-statements

Executes code block only when
if, *elif*, and *else* conditions are met

```
a = 1
b = 2

# execute each code block if condition is True
if a == b:
   # if a equals b
   print('a is equal to b’)
   b += 1
elif a < b:
   # else if a less than b
   print('a is less than b’)
   a -= 1
else:
   # otherwise, if a does not equal b
   # and if a is not less than b
   print('a is greater than b’)
   a *= b

c = a + b
print(c)
```

Boolean operators return True if condition(s) are met and False otherwise

```
7 == 6        # is-equal              (False)
7 != 6        # is-not equal          (True)
9 < 3         # less-than             (False)
8 <= 9        # less-than-or-equal    (True)
7 > 6         # greater-than          (True)
4 >= 5        # greater-than-or-equal (False)

1 < 3 and 3<2  # AND operator         (False)
1<3 or 3<2     # OR operator           (True)
not 3<2        # NOT operator          (True)
```

Operator precedence

Operations are evaluated in their order of precedence

```
# operator precedence (high to low)
(...), [...], {key: value}, {...}    # 1. groups, tuples, lists, dict., sets
x[index], f(arguments), x.attribute  # 2. containers, functions, objects
**                                   # 3. exponent
-x                                   # 4. negation
*, /, //, %                          # 5. multiply/division
+, -                                 # 6. addition/substraction
<, <=, >, >=, !=, ==                 # 7. comparisons
is, not, in, is, is not              #    (cont'd)
not x                                # 8. boolean not
and                                  # 9. boolean and
or                                   # 10. boolean or
```

Use parentheses to adjust precedence

```
5 * 2 + 4 * 3          # 22
(5 * 2) + (4 * 3)      # 22 (same precedence)
(5 * 2 + 4) * 3        # 42
5 * (2 + 4 * 3)        # 70
5 * ((2 + 4) * 3)      # 90
```

Combining operators

Are the following comparisons True or False?
Solve by hand.

```
# create variables
a = 1
b = 3
c = 2.1

# True or False?
a + b < c * 2                    # comparison 1
b + c - 0.1 >= 3 * c             # comparison 2
b * (a + c) < b**2 + (a / 10)    # comparison 3
b % c > ((c + a) > b) or ((2*a) > c)    # comparison 4
```

for-loops over list elements

Executes code block while iterating over each element in a container

```
# create list
x = [ 'a', 'b', 'c', 'd', 'e' ]
# get list length
n = len(x)
# loop over each element in list
for i in x:
    # code block
    s = i + ' (? out of ' + str(n) + ')'
    print(s)

# done
print('...done!')
```

code

```
a (? out of 5)
b (? out of 5)
c (? out of 5)
d (? out of 5)
e (? out of 5)
...done!
```

output

for-loops over indices

The *range(n)* function creates a list of integers with values *[0, 1, ..., n-1]*

```
# create list
x = [ 'a', 'b', 'c', 'd', 'e' ]
# loop over each integer in range
for i in range(len(x)):
    # code block
    s = x[i] + ' (' + str(i+1)
    s += 'out of ' + str(len(x)) + ')'
    print(s)

# done
print('...done!')
```

```
a (1 out of 5)
b (2 out of 5)
c (3 out of 5)
d (4 out of 5)
e (5 out of 5)
...done!
```

output

for-loops over dictionary items

Iterate over *(key, value)* ***items*** in a dictionary

```
# create dictionary
x = {'a':1, 'b':2, 'c':3}
# loop over all items in dictionary, while
# storing key and value for each item
for key,value in x.items():
    # code block
    s = 'key = ' + str(key) + '; '
    s += 'value = ' + str(value)
    print(s)

# done
print('...done!')
```

code

```
key = a; value = 1
key = b; value = 2
key = c; value = 3
...done!
```

output

enumerate

The *enumerate(x)* function assigned pairs an index to each iterable element in the container: *(index, value)*

```
# create dictionary
x = [10, 20, 30]
# create loop
for i,v in enumerate(x):
   # code block
   s = 'iteration = ' + str(i) + '; '
   s += 'value = ' + str(v)
   print(s)

# done
print('...done!')
```

code

```
iteration = 0; value = 10
iteration = 1; value = 20
iteration = 2; value = 30
...done!
```

output

Nested containers and loops

```
# create array of input
x = [[ 1,  4,  9],
    [16, 25, 36],
    [49, 64, 81]]

# create empty array for output
y = []

# iterate over rows
for i,row in enumerate(x):
  # create empty row for results
  y.append([])
  # iterate over column-values
  for j,val in enumerate(row):
    # get square root of input value
    y_ij = int( val**(1/2) )
    # store result in y[i][j]
    y[i].append(y_ij)

# print square roots
print(y)
```

code

Contains may be *nested* as elements within larger containers

For-loops may also be *nested* to process all containers, subcontainers, etc.

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

output

Test for element in container

The test "*x in y*" returns True if an element in *y* equals the value of *x*

```
>>> x = [ 1, 2, 3 ]
>>> y = 1
>>> if y in x:
...   print(f'{y} is in {x}')
...
1 is in [1, 2, 3]
```

using *in* test with integer list

```
>>> x = 'turducken'
>>> y = 'duck'
>>> if y in x:
...   print(f'{y} is in {x}')
...
duck in turducken
```

using *in* test with string

List concatenations

Use the + operator to ***concatenate*** lists with lists, or strings with strings

```
>>> turkey = 'gobble'
>>> duck = 'quack'
>>> turkey + duck + 'bock' 'gobblequackbock'
>>> x = [ 1, 2 ]
>>> y = [ 3, 4 ]
>>> x + y + [ 5, 6 ]
[1, 2, 3, 4, 5, 6]
```

Merge dictionaries

Merge the items in dictionary *y* into dictionary *x* using the *x.update(y)* method

```
>>> x = { 'cat':'meow', 'dog':'woof' }
>>> y = { 'cow':'moo', 'horse':'neigh' }
>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
>>> x.update(y)
>>> x
{'cat': 'meow', 'dog': 'woof', 'cow': 'moo', 'horse': 'neigh'}
```

List comprehensions

List comprehensions iterate through each element in a container using a compact notation; returns the processed list

```
>>> # example with list
>>> x = [ 1, 2, 3, 4, 5 ]
>>> # simple list comprehension
>>> y = [ (i**2) for i in x ]
[1, 4, 9, 16, 25]
>>> # list comprehension with if-statement
>>> z = [ (i**2) for i in x if i > 3 ]
[16, 25]
>>> # list comprehension for dictionary
>>> d = {'a':1, 'b':2, 'c':3}
>>> [ f'key={k},val={v}' for k,v in d.items() ]
['key=a,val=1', 'key=b,val=2', 'key=c,val=3']
```

Unpacking lists

Unpack a list to pass *x* as function arguments;
*f(*x)* will treat *x[0]* as arg1, *x[1]* as arg2, *etc.*

```
>>> def add(a,b):
...     return a + b

>>> x = [ 1, 2 ]

>>> # do not unpack `x`
>>> add(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() missing 1 required positional argument: 'b'

>>> # unpack `x` with `*x`
>>> add(*x)
3
```

Zipped containers

Use *zip(x,y)* to create a ***zipped container***
in which *z[i] = (x[i], y[i])*

```
>>> x = [ 'a', 'b', 'c' ]
>>> y = [ 1, 2, 3 ]
>>> zip(x,y)
<zip object at 0x7f3da6b93880>
>>> list(zip(x,y))
[('a', 1), ('b', 2), ('c', 3)]
>>> for i,j in zip(x,y):
...     print(f'zipped pair {i} and {j}’)

zipped pair a and 1
zipped pair b and 2
zipped pair c and 3
```

Index slicing, revisited

The notation *x[i:j:k]* retrieves elements *x[i]* through *x[j-1]* by every *kth* element

```
>>> x = list(range(10))
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> x[3:8:2]
[3, 5, 7]
>>> x[8:3:-2]
[8, 6, 4]
```

slicing list of integers

```
>>> y = 'syzygy'
>>> y[1:3]
'yz'
>>> y[::2]
'szg'
>>> y[::-2]
'yyy'
```

slicing a string

Overview for Lab 14