CONTROL FLOW

DATA 601
Python Basics

MULTI-LINE STATEMENTS

We have multiple ways of writing multi-line statements

Method I: Using the line continuation character (\) back-slash

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

Method II: Using parentheses ( ), brackets [ ] and braces { }.

```
a = (1 + 2 + 3 +
    4 + 5 + 6 +
    7 + 8 + 9)
```

```
colors = ['red',
          'blue',
          'green']
```

INDENTATION

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with **indentation** and ends with the first **un-indented** line.

The amount of indentation is up to you, but it must be consistent throughout that block.

We'll see examples on indentation in the following slides

CONTROL FLOW STATEMENTS

Branching:	If-this-do-this.
Looping:	Do this many times with slight differences.
Procedure calls:	Do this with some new information, and return an answer.
Sockets:	Read/write code from/to somewhere else.
Objects:	Make a toolkit to do a specific job.
Libraries:	Group toolkits for easy use.

IF COMPOUND STATEMENT

Compound statements are one or more clauses, the inside of which *must* be indented if not all on one line.

```
if condition:                    # Clause header.
        # do this                # Suite of statements.
        # do this
This line always done
```

or (rarer):

```
if condition: do this; do this
This line always done
```

EXAMPLE

```
if a < 10:
      print("a less than 10")
print("a assessed")
```

or

```
if (a < 10):
      print("a less than 10")
print("a assessed")
```

Parentheses not needed, but can make things clearer, especially with multiple operators. Again, don't rely on precedence.

LOGICAL OPERATORS IN PYTHON

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal
and	both must be true
or	one or both must be true
not	reverses the truth value

We'll talk about these in a few minutes

IF-ELSE

```
if condition:
        # do this
        # do this
else:
        # do this
        # do this
This line always done
```

IF-ELSE EXAMPLE

```
if a < 10:
        print("a less than 10")
else:
        print("a greater than 10")
print("a assessed")
```

THE IF-ELSE-IF LADDER

```
if condition-1:
          # do this
          # do this
elif condition-2:
          # do this
          # do this
elif condition-3:
          # do this
          # do this
else:
          # do this
          # do this
This line always done
```

IF-ELSE-IF EXAMPLE

```
if day <= 5:
        print("Weekday")
elif day == 6:
        print("Saturday")
else:
        print("Sunday")
```

But you have to watch for inefficiencies.

Put the most probable to the top

NESTED COMPOUND STATEMENTS

```
if a < 10:
        if b < 10:
                print("a and b less than 10")
        else:
                print("b greater than 10, a less")
print ("a and b assessed")
```

Note that to avoid ambiguity as to what the `else` links to, you can't do this all on one line.

CONDITIONS

Best to think of these as having to evaluate to either True or False.

```
a = 2
if (a == 2):                    # True
if (a != 2):                    # False
if (a != 3):                    # True
if not (a == 2):                # False


a  = True
if (a):                  # True
if not (a):              # False
a = False
if (a):                  # False
if not (a):              # True
```

BOOLEAN OPERATORS

```
if (a == 2) or (b == 3):                # If a == 2 OR b == 3
if (a == 2) and (b == 3):               # If a == 2 AND b == 3
```

OR and AND can therefore shortcut if the first condition is respectively true (for OR) or false (for AND).

Although this is possible:

```
if not a is None:
```

do instead:

```
if a is not None:
```

Note that empty sequences are false, so this is recommended by the docs:

```
if not seq:
if seq:
```

CONDITIONAL QUIRKS

```
x < y < z          # Is (x < y) and (y < z).
x < y > z          # Is fine.
```

TERNARY OPERATOR

```
x if condition else y
```

Which means: x if condition; y if not condition.

For example:

```
a = 10
b = "less than 5" if a < 5 else "more than five"
print(b)
```

The Python FAQ gives this nice example:

```
x, y = 50, 25
small = x if x < y else y
```

EXERCISE: QUADRATIC EQUATION

Suppose we want to know if the solutions to the quadratic equation

$$ax^2+bx+c=0$$

are real, imaginary, or complex for a given set of coefficients a, b, and c.

Ask user to enter values (one at a time) for a, b, and, c.

SOLUTION: QUADRATIC EQUATION

```
a = float(input("What is the coefficient a?"))
b = float(input("What is the coefficient b?"))
c = float(input("What is the coefficient c?"))

d = b*b - 4.*a*c
if d >= 0.0:
    print("Solutions are real")
elif b == 0.0:
    print("Solutions are imaginary")
else:
    print("Solutions are complex")
print("Finished!")
```

RANDOM INTEGER GENERATION

For next example, we'll need to generate some integers.

```
from random import randint
# let's generate a number between 0 and 100.
for x in range(6):
  print(randint(0,101))
```

The output will be something similar to (Remember each time we run this code, we are likely to get different 6 numbers

EXERCISE

Write a code which randomly generates an integer from 0 and 100 and assigns a grade using the following formula

- If score > 80, grade = A
- If 60< score <= 80, grade = B
- If 50< score <= 60, grade = C
- If 40<= score <= 50, grade = D
- If score < 40, grade = F

```
x = randint(0,101)
if x>80:
  print('Exam Score:', x, 'and assigned grade is A')
elif x>60:
  print('Exam Score:', x, 'and assigned grade is B')
elif x>50:
  print('Exam Score:', x, 'and assigned grade is C')
elif x>=40:
  print('Exam Score:', x, 'and assigned grade is D')
else:
  print('Exam Score:', x, 'and assigned grade is F')
```

REPEATING CODE: WHY WE NEED LOOPS

We could repeat code we need more than once:

```
i = 1
print (i)
i += 1
print (i)
i += 1
print (i)                    #… stop when i == 9
```

But each line means an extra line we might make an error on.

WHILE LOOPS

Instead, we can loop through the same code multiple times:

```
i = 1
while (i < 10):
        print (i)
        i += 1
```

This is far less error prone.

EXERCISE

Assume our computer's password is "abc123"

Ask user to enter the password

If the password is correct, print "You may enter"

If the password is not correct, give them two more chances

If the password is still not correct, print "You have entered wrong password for 3 times"

WHILE EXAMPLE: PASSWORD

```
password = ''
trial_number = 0
while password != 'abc123' and trial_number<3:
    print('What is the password?')
    password = input()
    trial_number = trial_number+1
if trial_number==3:
  print("You have entered wrong password for 3 times")
else:
  print('You have entered the correct password. You may enter.')
```

INFINITE LOOPS

Watch you don't do this:

```
i = 1
while (i < 10):
        print(i)
i += 1
```

Note that sometimes you want an infinite loop:

```
while (True):
        # Check for user interaction.
```

To break out of an infinite loop, use `CTRL-C` or equivalent.

EXERCISE

Fine the largest number in 1 million divisible by 17

BREAK

A bit like the evil goto, but cleaner.
Break ends looping entirely:

```
# Fine largest number in 1 million divisible by 17
i = 1000000
while (i != 0):
        if (i % 17 == 0):
                break
        i -= 1
print (i)
```

Firstly we don't want to keep counting down, and secondly we don't want to do "a -= 1" before printing.

WHILE/BREAK EXERCISE: GUESSING GAME

Let's generate a random integer from 1 to 6. then ask the user to make a guess. If it is the right guess, let user know. Otherwise let's ask them for another try. But the user can make 5 guesses at most

```
print("The computer chosen an integer from 1 to 6 randomly.”)
print(Try to guess it. You have 5 chances.")
x = randint(1,6)
trial_number = 1
while trial_number < 6:
  a = int(input("Please enter an integer from 1 to 6: "))
  if a==x:
    print("You guessed it correctly. Randomly chosen interger was", x)
    break
  else:
    trial_number = trial_number+1
if trial_number == 6:
  print("The chosen number was ", x)
```

CONTINUE

Continue ends current loop and starts next:

```
# Sum all even numbers in 1 million
i = 1000001
sum = 0
while (i != 0):
        i -= 1
        if (i % 2 == 1):
                continue
        sum += i
print (sum)
```

This often goes some way to making the code easier to read when the alternative is complicated nested if statements

WHILE-ELSE

It isn't commonly used, but you can put an "else" statement after while. This happens when the loop ends (which may be the first time), but not when the loop is broken out of. Essentially, it mashes up a loop and an if statement.

```
while (current_count < estimate_file_count):
        if file_not_found(current_count + ".txt"):
                print ("less files than expected")
                break
        current_count += 1
else:
        print (estimate_file_count + "files read")
```

COUNTING LOOPS

What if we want to count? We could do this:

```
i = 0

while (i < 10):

        print(i)

        i += 1
```

However, there are lots of mistakes we could make here.

```
ii = 0
while (i <= 10):
        print(a)
        j += 1
```

OTHER "FOR LOOPS"

Because of this, lots of languages have a 'for loop' construction, which places all these elements in one area, where they are clearly related and can't be lost.

```
for (int i = 0; i < 10; i++) {
   ...
}
```

Java example

PYHTON FOR-LOOPS

Python takes a different approach. Python works with sequences, that is, you give it an object containing the numbers, and it works through them one at a time.

```
for loop_control_target_variable in sequence:
        # do this
```

Example:

```
for i in (1,2,3,4,5,6,7,8,9):
        print(i)
```

PYTHON FOR LOOPS

This may seem strangely verbose, but it is very powerful. It means you can do this:

```
for name in ("Dale", "Albert", "Gordon", "Tamara"):
        print(name)
```

Moreover, the syntax will take a sequence object:

```
names = ("Dale", "Albert", "Gordon", "Tamara")
for name in names:
        print(name)
```

You can imagine, for example, reading the names in from a file.

ITERATORS

- In fact, what is happening is that for loops work with a type of construct called an iterator.
- Pretty much all sequences are iterable (that is, there's a "next" object) and have a function to get an iterator object representing themselves.
- The iterator has a function that gives the for loop the next object in the sequence when asked.
- This doesn't especially matter here, but the terminology is used a lot in the documentation and examples.
- You can generate an iterator from a sequence yourself with:

```
a = iter(sequence)
```

RANGE AND SLICES

As both ranges and slices can be treated as sequences, we can do this:

```
for i in range(10):                    # i = 0,1,2,3,4,5,6,7,8,9

for i in range(5,10):                  # i = 5,6,7,8,9

for i in range(5,10,2):                # i = 5,7,9


names = ("Dale","Albert","Gordon","Tamara","Philip","Chester","Windom")

for name in names[1:5:2]:              # name = "Albert", "Tamara"
```

INDICES

To combine having a number and an object it indexes, we can do this:

```
names = ("Dale","Albert","Gordon","Tamara","Philip","Chester","Windom")
for i in range(len(names)):
        print(i, names[i])
```

However, you cannot change i during this to skip objects:

```
for i in range(len(names)):
        print(i, names[i])
        i += 1
```

Assignment creates a new temporary variable, and the target variable resets to its next value at the top of the loop - it is coming from the iterator. This creates some issues…

EFFICIENCY

Slices copy containers, while ranges are iterators that actually generate the values one at a time. It is better, therefore, with long sequences to do:

```
names = ("Dale","Albert","Gordon","Tamara","Philip","Chester","Windom")

for i in range(2,len(names),2):

        print(names[i])
```

Than

```
for name in names[2::2]:
        print(name)
```

MODIFYING LOOP SEQUENCES

The one disadvantage of not having an index counter is that it makes it hard to remove items from a list.

Usually, if you add or remove something, you'd increase or decrease the loop counter to accommodate the change.

As you can't adjust the loop control target variable, you can't do this.

EXAMPLE

```
# Mutable list

names_tuple =['Dale', 'Albert', 'Gordon', 'Tamara', 'Philip', 'Chester', 'Windom']

for name in names_tuple:

        names_tuple.remove(name)

print(names)
```

May think this removes all names, but in fact, this happens:
i) internal index 0, Dale removed; Albert goes to position 0, Gordon to position 1, etc.
ii) internal index goes to 1, now references Gordon, Gordon removed not Albert, Tamara moves to position 1…
Output is: Albert, Tamara, Chester, not nothing

SOLUTION

For Python, it is recommended you loop through a copy of the list; you can copy a list with a complete slice.

```
names = ["Dale","Albert","Gordon","Tamara","Philip",
                                   "Chester","Windom"]
for name in names[:]:              # Note full slice
        names.remove(name)
print(names)
```

This way you traverse every object in the copy, without missing any, because you're removing from a different list.

BREAK

If a for loop is terminated by break, the loop control target keeps its current value.

Again, you can add an extra else clause not visited when break acts.

```
for target in sequence:

else:
```

2D LOOPS

To loop through two dimensions, nest loops:

```
data = [
[0,1,2],
[3,4,5]
]


for row in data:
    for item in row:
        print (item)
```

2D LOOPS

However, it is often necessary to know the coordinates in collection space of the item you're referencing.

```
data = [
[0,1,2],
[3,4,5]
]


for i in range(len(data)):
    for j in range(len(data[i])):
        print (data[i][j])
```

NESTING LOOPS

```
for i in range(len(data)):
    for j in range(len(data[i])):
        print (data[i][j])
```

Variables re-made if a loop runs more than once.

1) The outer loop starts, then the inner loop starts.
2) When the inner loop has run once, it returns to the start of the inner loop, not the outer loop.
3) It keeps doing this until run to completion (j == len(data[i]); i still zero).

What do you think happens to j then, and where does the code go next?

NESTING LOOPS

```
for i in range(len(data)):
    for j in range(len(data[i])):
        print (data[i][j])
```

4) j is destroyed, and the outer loop increments to i = 1.

5) The inner loop runs again, j recreated as j = 0.

6) The inner loop runs to completion.

Thus, each time the outer loop runs once, the inner loop runs to completion.

7) This is repeated until the outer loop completes.

8) The code then moves on.

NESTED LOOPS

```
data = [
[0,1,2],
[3,4,5]
]


for i in range(len(data)):
    for j in range(len(data[i])):
        print (data[i][j])
```

Let's look at i and j:

i	j
0	0
0	1
0	2
1	0
1	1
1	2

This is exactly what we need for moving down a row at a time in our array (i) and then running across each row a space at a time (j).

EXAMPLE

We want to print how many characters are in the following tv_shows, i. e.

```
The title ABC is 3 characters long.
```

```
tv_shows = [['How I Met Your Mother', 'Friends', 'Silicon Valley'],
    ['Family Guy', 'South Park', 'Rick and Morty'],
    ['Breaking Bad', 'Game of Thrones', 'The Wire']]
```

```
for sublist in tv_shows:
    for show_name in sublist:
        char_num = len(show_name)
        print("The title " + show_name + " is " + str(char_num) + " characters long.")
```

```
The title How I Met Your Mother is 21 characters long.
The title Friends is 7 characters long.
The title Silicon Valley is 14 characters long.
The title Family Guy is 10 characters long.
The title South Park is 10 characters long.
The title Rick and Morty is 14 characters long.
The title Breaking Bad is 12 characters long.
The title Game of Thrones is 15 characters long.
The title The Wire is 8 characters long.
```

ISSUES

This is surely one of the neatest algorithms ever!

However, it is easy to make mistakes which are avoided by directly using the target variable to access items.

There are three problems with the below. Can you spot them?

```
for i in range(len(data)):
    for i in range(len(data[j])):
            data[j][i] = 10
```

2D ISSUES

The three mistakes are classics that everyone makes, even experienced coders:

```
for i in range(len(data)):
    for i in range(len(data[j])):
            data[j][i] = 10
```

`len(data[j]))=>` Looping through to the wrong dimension length.
This is very common if the lengths are hard-wired in, so avoid that.

`for i` Cutting and pasting your outer loop to make your inner loop, and forgetting to change part of the variable use; here, the inner increment should be to j.

`data[j][i]` Switching the indices the wrong way round. This should be `data[i][j]`. With an non-square array, this will result in trying to read off one side of the array and the program will break. Worse, with a square array, your data will silently be transposed.

PRINT

Print inserts spaces when comma separated.

By default ends with a newline. But this can be overridden (best in scripts):

```
print ("a", end=",")
print ("b", end=",")
print ("c")
```

```
a,b,c
```

WHEN TO ACT

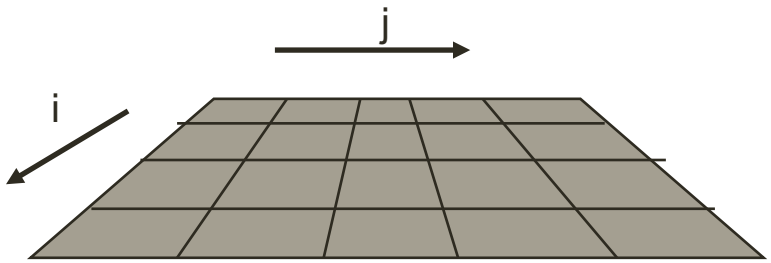
```
for i in range(len(data)):
        # Line here done every outer loop
    for j in range(len(data[i])):
                # Line here done every inner loop
                data[i][j] = 10
        # Line here done every outer loop
```

```
for i in range(len(data)):
        for j in range(len(data[i])):
                print (data[i][j], end=",")
        print ("")
```

MOVING WINDOW ALGORITHMS

A prime example of why we might want the coordinates is moving window algorithms. Let's start with a simple allocation to the current item:

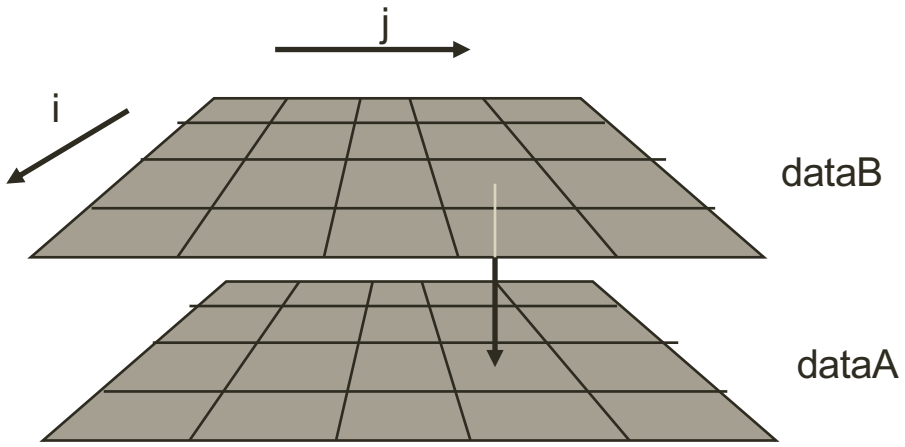
```
for i in range(len(data)):
    for j in range(len(data[i])):
                data[i][j] = 10
```



VARIATIONS

Looping through the same positions in two collections:

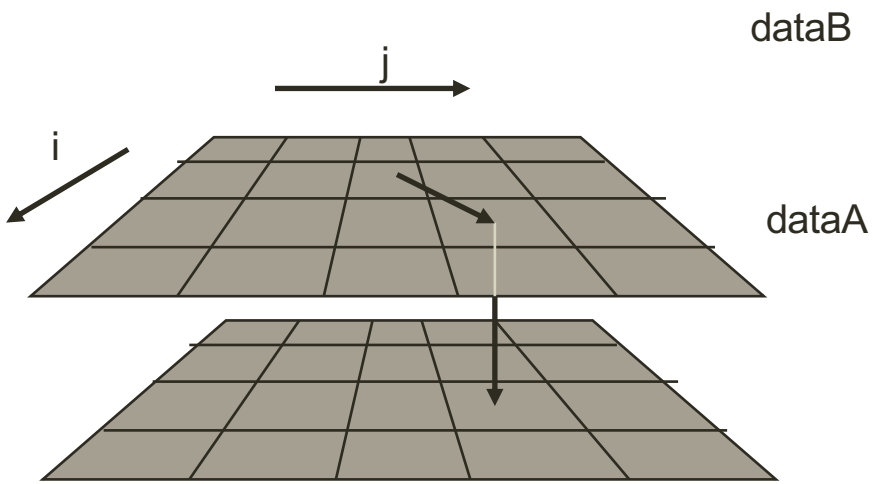
```
for i in range(len(dataA)):
    for j in range(len(dataA[i])):
            dataA[i][j] = dataB[i][j]
```



VARIATIONS

Looping through two arrays at positions relative to one array (note boundary problem):

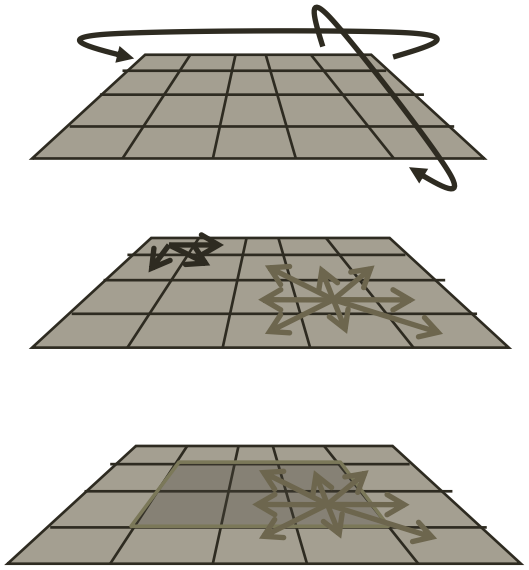
```
for i in range(len(dataA)):
    for j in range(len(dataA[i])):
                dataA[i][j] = dataB[i-1][j-1]
```



BOUNDARY PROBLEMS

Various solutions.

Depends on problem context.



Wrap boundaries:
Suitable for modelling abstract landscapes

Only process as many cells as you can:
Suitable for modelling non-abstract landscapes

Only process cells that can have complete processing:
Suitable for image processing

MULTIPLE TARGETS

If you know the number of elements in the second dimension of a 2D list or tuple, you can:

```
data = [
[0,1],
[2,3],
[4,5]
]
for a,b in data:
        print (str(a) + " " + str(b))
```

a ➔ Column 0 b ➔ Column 1

ZIP USES

Really useful for looping through two sets of data at the same time.
Here, for example, we use it to make a dictionary from two lists:

```
a = ['fname','lname','byear','state','zipcode']

b = ['mike','simsek','1971','md','20817']

z = zip(a,b)

d = {}

for t in z:

        d[t[0]] = t[1]

print(d)

% OUTPUT: {'fname': 'mike', 'lname': 'simsek', 'byear':
'1971', 'state': 'md', 'zipcode': '20817'}
```

zip():
map the similar index of multiple containers so that they can be used just using as single entity.

BUILTINS: ITERATORS

next() returns the next item from the iterator.

```
next(iterator[, default])
```

this value is returned if the iterator is exhausted (no items left)

If you have any doubts about the iterator returning a value, this will return a default value at the end of the iterator. Obviously make sure it doesn't create an infinite loop.

If the default isn't given, it produces a warning at the end.

```
a = list(range(3))


it = iter(a)
for i in range(5):
    print(next(it, "missing"))
```

```
0
1
2
missing
missing
```

REVERSED(SEQUENCE)

```
reversed(seq)
```

Get a reverse iterator.

```
a = list(range(3))
ri = reversed(a)
for i in ri:
    print(i)
```

SORTING LISTS

```
a.sort()        # Sorts list a. From then on, the list is sorted.
b = sorted(a)   # Copies list a, sorts it, and attaches the copy to b.
```

The former is more memory efficient. The latter leaves the original unchanged and will work on tuples, as it generates a list.

There are a wide variety of options for sorting, including defining the kind of sorting you want. See:

https://docs.python.org/3/howto/sorting.html