Pseudo-code and Iterations

Alexander Lam

Wong Tai Sin

- At Wong Tai Sin temple, you can draw lots to get your fortune.
  - Bamboo sticks labelled from 1 to 100
  - Give your name, address, birthday, birthplace, and ask the Gods a question
  - https://www.pswu.com/wongtaisin/
  - For some reason, Wong Tai Sin temple needs your help to generate a random list of unique numbers from 0 to 99.

Pseudo-code

- Generate a random list of unique numbers from 0 to 99

```
Repeat
    Generate first digit 0 to 9 using a 10-faced die
    Generate second digit 0 to 9 using a 10-faced die
    If number already exists, try again
    If number does not exist, output to screen or file
Until all 100 numbers are generated
```

Pseudo-code

- Pseudo, coming from the Greek word ψευδής (*pseudés*), means false.
- Pseudo-code refers to high level program code being expressed in English
  - Reflects lines of thought
  - Higher level than program statements
- Example

  - Rearrange the employees in a random order.
  - Check if the employees are correctly ordered by their birthdays.
  - If yes, we are done.
  - If no, repeat the program.

Pseudo-code

- Pseudo, coming from the Greek word ψευδής (*pseudés*), means false.
- Pseudo-code refers to high level program code being expressed in English
  - Reflects lines of thought
  - Higher level than program statements
- Example

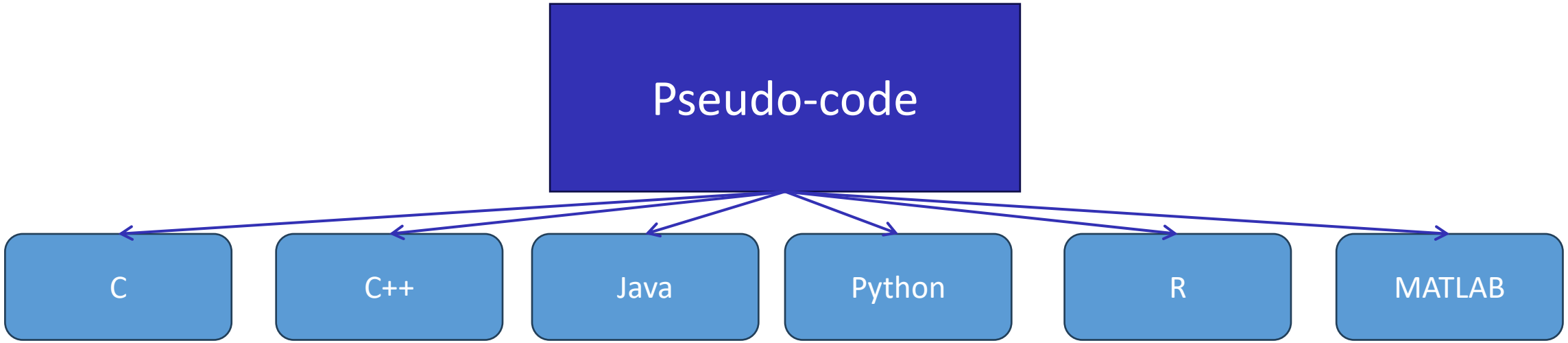
  - Take the list L of employees and their respective birthdays.
  - Repeat
    - For i in [0..length(L)-2]
      - Compare the employees in L[i] and L[i+1]. If they are in the wrong order, swap them.

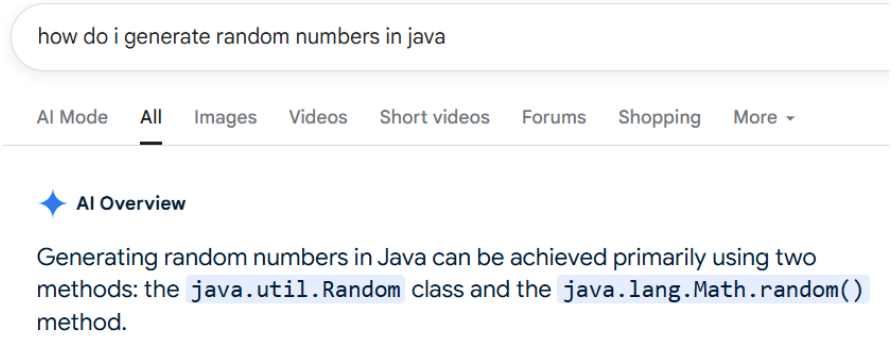
  until list is properly sorted.

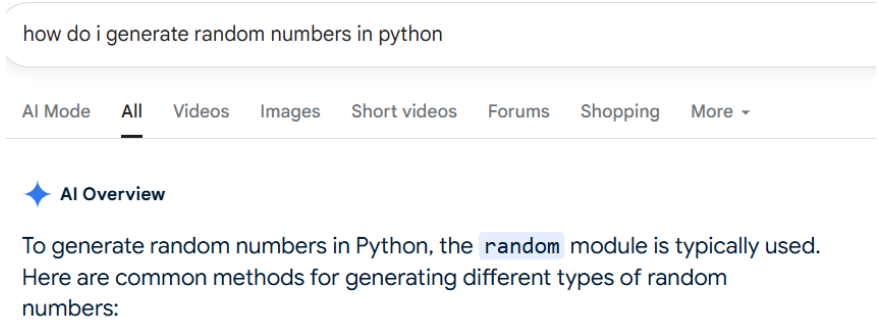
- More details on this about 2 weeks!

Importance of Learning Pseudo-code

- Knowledge of pseudo-code and computational problem solving is the foundation to learning any programming language.

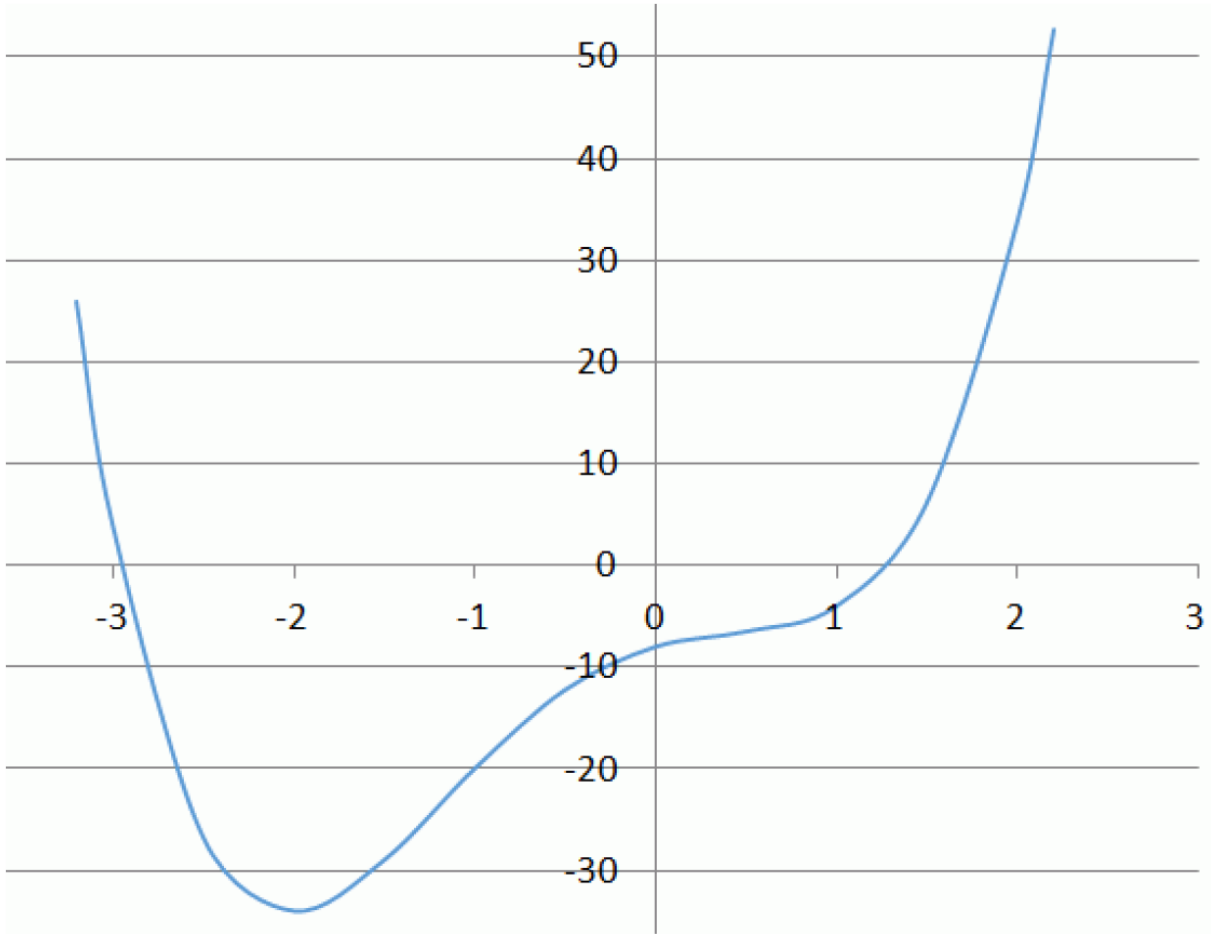






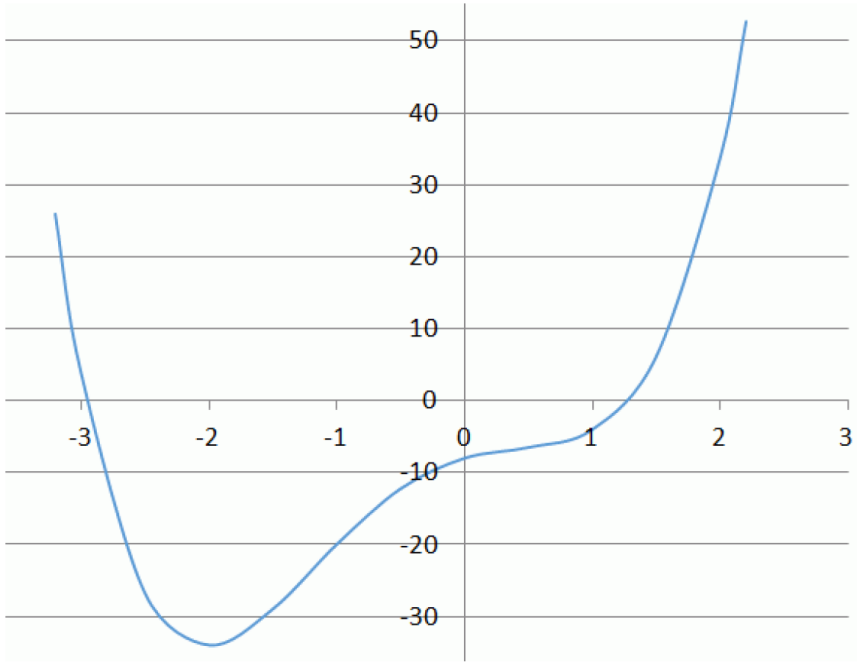
Pseudo-code - A problem

- Solve $f(x) = 2x^4 + 3x^3 - 6x^2 + 5x - 8 = 0$



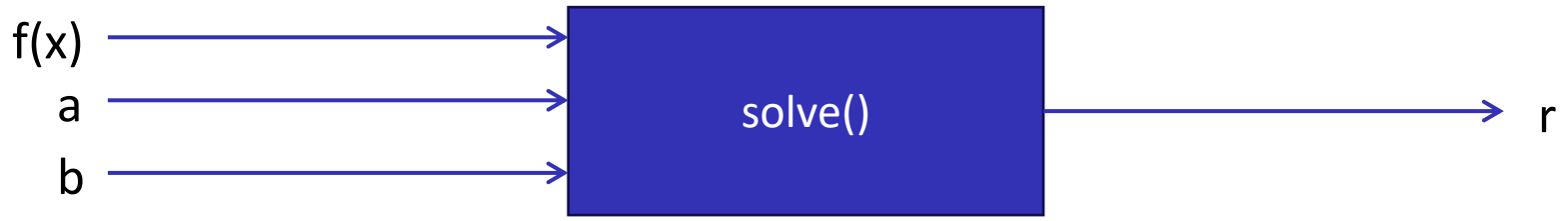
Pseudo-code – A problem

- Solve $f(x) = 2x^4 + 3x^3 - 6x^2 + 5x - 8 = 0$
- **Theorem:** If a function $f(x)$ is continuous between a and b, and $f(a)$ and $f(b)$ have opposite signs, then there exists a root between a and b.



Pseudo-code – Our solution

- **Input:** continuous mathematical function f(x), a and b such that a<=b.
- **Output:** a root r **between** (not including) a and b such that f(r) is sufficiently close to zero.



Pseudo-code – Our Solution

- **Input:** continuous mathematical function f(x), a and b such that a<=b.
- **Output:** a root r **between** (not including) a and b such that f(r) is sufficiently close to zero.
- Check whether f(a) and f(b) have different signs
  - If they have the same sign, return "f(a) and f(b) have the same sign"
- Set left = a, right = b # root between left and right
- Repeat until root is found
  - Set mid = (left+right)/2
  - If f(mid) is sufficiently close to zero, return r = mid.
  - If f(left) and f(mid) have opposite signs, then
    - Set right = mid
  - Otherwise, set left = mid

Pseudo-code - Questions

- Is the program correct?
  - If not, where is the problem?
- Why sufficiently close to zero and not equal to zero?
  - How to implement “sufficiently close to zero”?
- Can we remove the a and b inputs, and tell the program to find them instead?

Pseudo-code – Summing a list of numbers

- **Input:** a list of numbers, L
  - E.g. L = [1,5,8,2,7,9]
- **Output:** the sum S of the numbers in L
- **Pseudo-code:**
  - Set S = 0 <- Can we skip this step?
  - For each number n in list L
    - Add n to S
  - Return S

Pseudo-code – Multiplying a list of numbers

- **Input:** a list of numbers, L
  - E.g. L = [1,5,8,2,7,9]
- **Output:** the product P of the numbers in L
- **Pseudo-code:**
  - Set P = 1
  - For each number n in list L
    - Multiply P by n
  - Return P

Pseudo-code – Looking up a Term in a Dictionary

- **Input:** a term t to look up, a dictionary
- **Output:** a dictionary explanation E of term t
- **Pseudo-code:**
  - Set p=1
  - Until t is found
    - Turn to page p and look for t
    - If t is found on page p, return its explanation E
    - Otherwise, increment p

Pseudo-code – Looking up a Term in a Dictionary

- **Input:** a term t to look up, an (ordered) dictionary
- **Output:** a dictionary explanation E of term t
- **"Better" Pseudo-code:**
  - Set start = 1 and end = last page
  - Until t is found or end <= start
    - Set mid = (start+end)/2
    - Check if page mid has term t
      - If yes, return corresponding explanation E
      - If term t is before first word in page mid, set end = mid # first part
      - Otherwise, set start = mid # second part
  - If t is not found, return "not found"

Why do we need this?

Pseudo-code – Depth of detail

- Specifically how are we looking up term t in each page?
  - Top to bottom?
  - Bottom to top?
  - “Mid splitting” idea?
- We would need to add this extra detail into our pseudo-code
- Pseudo-code can have different levels of detail
  - We typically start at a very high level and refine it towards a lower level
  - Program code is at the lowest level of detail
  - Ideally, ‘finished’ pseudo-code should be able to be accurately programmed by an intern

Pseudo-code – Looking up a Term in any book

- What if we want to find the term “iteration” in a Python textbook?
- ‘Split-by-middle’ approach no longer works

- Back to sequentially searching from start to finish?
  - Any faster alternatives?

Recall

- Rearrange the employees in a random order.
- Check if the employees are correctly ordered by their birthdays.
- If yes, we are done.
- If no, repeat the program.

- This pseudocode describes the computational steps forming the solution to a problem.
- We can ‘execute’ pseudocode with the help of pen and paper.
- Computers can execute program code derived from pseudocode.

Computational Steps

- Rearrange the employees in a random order.
- Check if the employees are correctly ordered by their birthdays.
- If yes, we are done.
- If no, repeat the program.

- How many types of computational steps?
  - Repeat until [employees correctly ordered]
    - Iterative
  - If [employees are correctly ordered]
    - Conditional
  - Step-after-step execution
    - Sequential

Computational Steps

- There are three types of computational steps
  - Sequential: statements executed from top to bottom
    - E.g. arithmetic expression (x = 2*i)
    - E.g. input or output statement (print(“Hello World!”))
  - Conditional: Action taken depending on whether condition is true or false
  - Iterative: Part of the code which is repeated, also called a loop
    - Definite iteration: Iteration is repeated for a specific, known number of times.
    - Indefinite iteration: It is not exactly clear how many times the iteration is repeated.

Definite and Indefinite Iterations

- Definite Iteration

```
Sum = 0
for i in [1..4] do
    Sum = Sum + i
return Sum
```

```
P = 1
for each number n in list L
    multiply P by n
return P
```

- Indefinite Iteration

```
j = 1
repeat
    print j
    j = j + 1
until j > 4
```

- Rearrange the employees in a random order.
- Check if the employees are correctly ordered by their birthdays.
- If yes, we are done.
- If no, repeat the program.

Computational Steps

- Computational Steps: Sequential, Conditional, and Iterative
- Any more types of steps?
  - Arguably yes but…
- These three types of steps are sufficient to express any program.
  - We say that they are “functionally complete”.
    - E.g. sine and cosine are sufficient to express the remaining 4 trigonometric functions tangent, cotangent, secant, and cosecant. Even cosine can be expressed using sine!
  - Other types of computational steps exist to make program development easier and more convenient.
  - A function is one major type that helps reduce repetition when writing code.

Iterations

- Iterations are often called loops.
- E.g. Definite iteration

```
for i in [1..4] do
    print i
```

- How many times is this executed?
- Indefinite iteration

```
j = 1
repeat
    print j
    j = j + 1
until j > 4
```

- How many times is this executed?

range(1,5) means
1 to 4 in Python

```
for i in range(1,5):
    print(i)
```

range(a,b) means
a to b-1 in Python

Definite vs Indefinite iterations

- Definite iterations are easier, it is **very clear** how many times the program loops the code.
  - The looping variable is simple.
- Indefinite iterations are harder, need to **think carefully** about how many times the programs loops the code.
  - Looping variable can be more complicated.
- The difference can sometimes be blurry and subjective.
  - In general, for loops **tend** to be definite, and **repeat until** loops tend to be indefinite.

Definite vs Indefinite iterations

- In general, for loops **tend** to be definite, and **repeat until** loops tend to be indefinite.
  - For loops are sequential
  - Repeat until loops are conditional

```
for i in [1..4] do
    print i
```

```
j = 1
repeat
    print j
    j = j + 1
until j > 4
```

Correctness of Iterations

- How do we know our iterations (esp. indefinite iterations) are correct?
  - The total number of executions should be correct.
  - The values produced in each iteration should be correct.
- Pseudocode:
  - Very careful analysis
  - Try to program the pseudocode
- Program code:
  - My favourite debugging strategy: Lots of print statements

Loop Unfolding

- We can unfold a loop to see behavior better.
  - Definite loop

```
for i in [1..4] do
    print i
```

```
i = 1
print i
i = 2
print i
i = 3
print i
i = 4
print i
```

Loop Unfolding

- Unfolding an indefinite loop

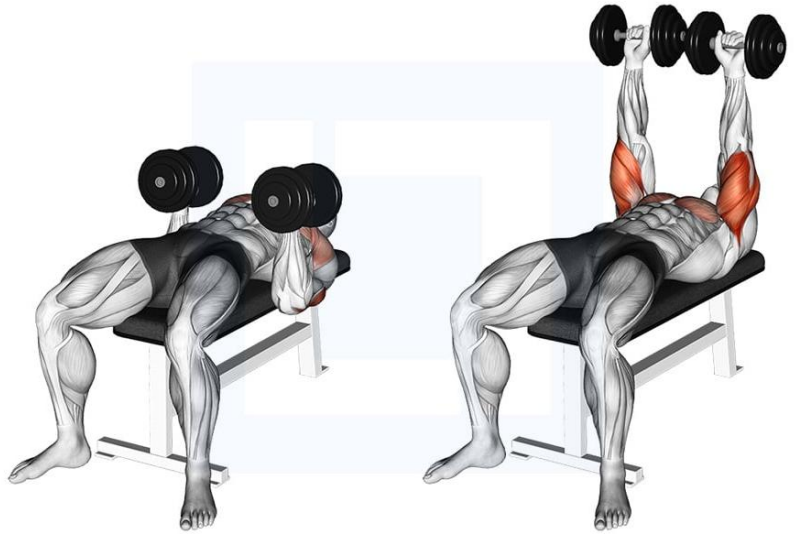
```
j = 1
repeat
    print j
    j = j + 1
until j > 4
```

```
j = 1
print j
j = j + 1
if j > 4 then stop repeating
print j
j = j + 1
if j > 4 then stop repeating
print j
j = j + 1
if j > 4 then stop repeating
print j
j = j + 1
if j > 4 then stop repeating
. . .
```

Gym Exercises

- When doing a gym exercise, you perform the movement for several **repetitions (reps)** before resting.
- Each collection of consecutive reps is called a **set**.
- You typically count upwards to keep track of your reps in each set.
- Alex is old and tired, and can only do 4 reps in each set.
- Also, he is having trouble counting and using definite loops
- Let's help him count to 4 reps using indefinite loops!

```
print(1)
1
print(2)
2
print(3)
3
print(4)
4
```



Iterations

- Indefinite iteration 1

```
j = 1
repeat
   print j
   j = j + 1
until j > 4
```

- How many times is it executed?
- What is printed?

- Indefinite iteration 2

```
j = 1
repeat
   print j
   j = j + 1
until j >= 4
```

- How many times is it executed?
- What is printed?

Iterations

- Indefinite iteration 3

```
j = 0
repeat
   j = j + 1
   print j
until j > 4
```

- How many times is it executed?
- What is printed?

- Indefinite iteration 4

```
j = 0
repeat
   j = j + 1
   print j
until j >= 4
```

- How many times is it executed?
- What is printed?

Iterations

- Indefinite iteration 5

```
j = 1
while j <= 4 do
   print j
   j = j + 1
```

- How many times is it executed?
- What is printed?

- Indefinite iteration 6

```
j = 1
while j < 4 do
   print j
   j = j + 1
```

- How many times is it executed?
- What is printed?

Iterations

- Indefinite iteration 7

```
j = 0
while j <= 4 do
   j = j + 1
   print j
```

- How many times is it executed?
- What is printed?

- Indefinite iteration 8

```
j = 0
while j < 4 do
   j = j + 1
   print j
```

- How many times is it executed?
- What is printed?

Iterations

		Once too few	Once too many	
	1/5	**2/6**	**3/7**	**4/8**

Program	j = 1
	repeat
	  print j
	  j = j + 1
	until j > 4
		j = 1
		repeat
		  print j
		  j = j + 1
		until j >= 4
			j = 0
			repeat
			  j = j + 1
			  print j
			until j > 4
				j = 0
				repeat
				  j = j + 1
				  print j
				until j >= 4

Output	1 2 3 4	1 2 3	1 2 3 4 5	1 2 3 4

Program	j = 1
	while j <= 4 do
	  print j
	  j = j + 1
		j = 1
		while j < 4 do
		  print j
		  j = j + 1
			j = 0
			while j <= 4 do
			  j = j + 1
			  print j
				j = 0
				while j < 4 do
				  j = j + 1
				  print j

Output	1 2 3 4	1 2 3	1 2 3 4 5	1 2 3 4

While vs Repeat-until

- While loops keep iterating while the condition is true, and stop when it is false
  - Most languages (including Python) use While loops
- Repeat-Until loops keep iterating while the condition is false, and stop when it is true
  - Pascal and Visual Basic use Repeat-Until loops

Iterations

	Once too many	Once too few	Off by one	Off by one
	9/13	**10/14**	**11/15**	**12/16**

Program	j = 0
	repeat
	  print j
	  j = j + 1
	until j > 4
		j = 0
		repeat
		  print j
		  j = j + 1
		until j >= 4
			j = 1
			repeat
			  j = j + 1
			  print j
			until j > 4
				j = 1
				repeat
				  j = j + 1
				  print j
				until j >= 4

Output	0 1 2 3 4	0 1 2 3	2 3 4 5	2 3 4

Program	j = 0
	while j <= 4 do
	  print j
	  j = j + 1
		j = 0
		while j < 4 do
		  print j
		  j = j + 1
			j = 1
			while j <= 4 do
			  j = j + 1
			  print j
				j = 1
				while j < 4 do
				  j = j + 1
				  print j

Output	0 1 2 3 4	0 1 2 3	2 3 4 5	2 3 4

Iterations

	17/21	18/22	19/23	20/24

Program	j = 1
	repeat
	  print j
	  j = j + 1
	until j < 4
		j = 0
		repeat
		  print j
		  j = j + 1
		until j <= 4
			j = 1
			repeat
			  j = j + 1
			  print j
			until j < 4
				j = 0
				repeat
				  j = j + 1
				  print j
				until j <= 4

Output	1	0	2	1

Program	j = 1
	while j >= 4 do
	  print j
	  j = j + 1
		j = 0
		while j > 4 do
		  print j
		  j = j + 1
			j = 1
			while j >= 4 do
			  j = j + 1
			  print j
				j = 0
				while j > 4 do
				  j = j + 1
				  print j

Output	Nothing printed	Nothing printed	Nothing printed	Nothing printed

Iterations

- Issues to notice for indefinite loops:
  - Initial loop variable value
  - Loop termination conditions: `<` vs `<=` or `>` vs `>=` (sometimes `!=` vs `==`)
  - Moment loop variable value is changed
- Print loop variables to check:
  - At beginning of loop execution
  - At end of loop execution
  - At moment when value of loop variable is changed
  - At moment when value of loop variable is used (right value used)
- Count number of executions
- Human loop variable value checking when debugging on paper.
- Formal approach: program correctness verification, by means of loop-invariant analysis.

Iterations

```
j = 1
while j<= 4
    print j (to debug)
    j = j + 1
    print j (to debug)
    giveAlex(j dollars)
    print j (to debug)
    j = 1
    print j (to debug)
    j = j * 3
    print j (to debug)
    j = factorial(j)
    print j (to debug)
    giveAlex(j dollars)
    print j (to debug)
```

Debugging: Every time we touch the counter, use a print statement to keep track of it!

Iterations

- Comparing the use of < and <= in the conditions used in while and repeat loops.
  - It seems that using <= would have the loop executed one more time.
  - Is that true?

	1	2	7	8

Program	j = 1
	repeat
	  print j
	  j = j + 1
	until j > 4
		j = 1
		repeat
		  print j
		  j = j + 1
		until j >= 4
			j = 0
			while j <= 4 do
			  j = j + 1
			  print j
				j = 0
				while j < 4 do
				  j = j + 1
				  print j

Output	1 2 3 4	1 2 3	1 2 3 4 5	1 2 3 4

Iterations

- What are the outputs and number of times executed?

	1/1’	2/2’	7/7’	8/8’
Program	j = 1 repeat print j j = j + 1 until j > 4	j = 1 repeat print j j = j + 1 until j >= 4	j = 0 while j <= 4 do j = j + 1 print j	j = 0 while j < 4 do j = j + 1 print j
Output	1 2 3 4	1 2 3	1 2 3 4 5	1 2 3 4
Program	j = 1 repeat print j j = j + 2 until j > 8	j = 1 repeat print j j = j + 2 until j >= 8	j = 0 while j <= 8 do j = j + 2 print j	j = 0 while j < 8 do j = j + 2 print j
Output	1 3 5 7	1 3 5 7	2 4 6 8 10	2 4 6 8

Iterations

- **Note**: In Python, a while loop checks its condition before the first iteration, and **between** each iteration.
  - **Never checked within the loop body**

```
1   i=4
2   while i<5:
3       print(i)
4       i=i+1
5       print(i)
6       i=i+1
7       print(i)
8       i=i+1
9       print(i)
10      i=i+1
11      print(i)
12      i=i+1
```

```
4
5
6
7
8
```

Iterations

Instead of counting up, we can count down

- Definite Iteration

```
for i in [4..1] do
    print i
```

4,3,2,1 (not including 0)

```
for i in range(4,0,-1):
    print(i)
```

- Indefinite Iteration

```
j = 4
repeat
    print j
    j = j - 1
until j < 1
```

Iterations

- Unfold the indefinite loop

```
j = 4
repeat
    print j
    j = j - 1
until j < 1
```

```
j = 4
print j
j = j - 1
if j < 1 then stop repeating
print j
j = j - 1
if j < 1 then stop repeating
print j
j = j - 1
if j < 1 then stop repeating
print j
j = j - 1
if j < 1 then stop repeating
. . .
```

Iterations

		Missing a 1	Off by 1	Missing a 4
	1/5	**2/6**	**3/7**	**4/8**

Program	j = 4
	repeat
	  print j
	  j = j - 1
	until j < 1
		j = 4
		repeat
		  print j
		  j = j - 1
		until j <= 1
			j = 4
			repeat
			  j = j - 1
			  print j
			until j < 1
				j = 4
				repeat
				  j = j - 1
				  print j
				until j <= 1

Output	4 3 2 1	4 3 2	3 2 1 0	3 2 1

Program	j = 4
	while j >= 1 do
	  print j
	  j = j - 1
		j = 4
		while j > 1 do
		  print j
		  j = j - 1
			j = 4
			while j >= 1 do
			  j = j - 1
			  print j
				j = 4
				while j > 1 do
				  j = j - 1
				  print j

Output	4 3 2 1	4 3 2	3 2 1 0	3 2 1

Iterations

	9/13	10/14	11/15	12/16

Program	j = 4
	repeat
	  print j
	  j = j - 1
	until j > 1
		j = 4
		repeat
		  j = j - 1
		  print j
		until j > 1
			j = 4
			repeat
			  print j
			  j = j + 1
			until j < 1
				j = 4
				repeat
				  j = j + 1
				  print j
				until j < 1

Output	4	3	4 5 6 7 8 …	5 6 7 8 9 …

Program	j = 4
	while j <= 1 do
	  print j
	  j = j - 1
		j = 4
		while j <= 1 do
		  j = j - 1
		  print j
			j = 4
			while j >= 1 do
			  print j
			  j = j + 1
				j = 4
				while j >= 1 do
				  j = j + 1
				  print j

Output	Nothing printed	Nothing printed	4 5 6 7 8 …	5 6 7 8 9 …

Iterations

	17/21	18/22	19/23	20/24

Program	j = 4
	repeat
	  print j
	  j = j - 1
	until j < 1
		j = 4
		repeat
		  print j
		  j = j - 1
		until j <= 1
			j = 4
			while j >= 1 do
			  print j
			  j = j - 1
				j = 4
				while j > 1 do
				  print j
				  j = j - 1

Output	4 3 2 1	4 3 2	4 3 2 1	4 3 2

Program	j = 8
	repeat
	  print j
	  j = j - 2
	until j < 1
		j = 8
		repeat
		  print j
		  j = j - 2
		until j <= 1
			j = 8
			while j >= 1 do
			  print j
			  j = j - 2
				j = 8
				while j > 1 do
				  print j
				  j = j - 2

Output	8 6 4 2	8 6 4 2	8 6 4 2	8 6 4 2

Iterations

Note: Conditions to end loop are not limited to inequalities

```
rep = 1
repeat
    print rep
    rep = rep + 1
until alex is tired
```

```
rep = 1
While alex is not tired
    print rep
    rep = rep + 1
```

Iterations

- Iterations can be nested

```
for i in [1..3] do
    for j in [1..3] do
        print i, j
```

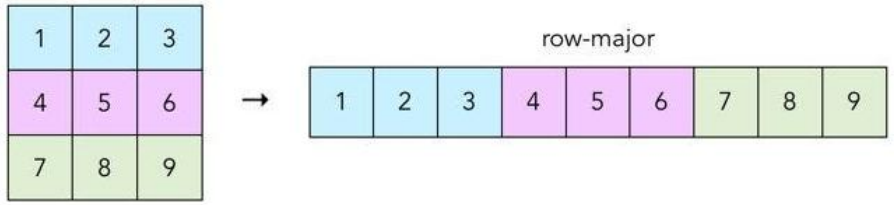
Unfold i

```
i = 1
for j in [1..3] do
    print i, j
i = 2
for j in [1..3] do
    print i, j
i = 3
for j in [1..3] do
    print i, j
```

$$A = \begin{bmatrix} 1,1 & 1,2 & 1,3 \\ 2,1 & 2,2 & 2,3 \\ 3,1 & 3,2 & 3,3 \end{bmatrix}$$

- This gives all combinations of i and j values (3 x 3 = 9 pairs), but what order?
  - 1 1 / 1 2 / 1 3 / 2 1 / 2 2 / 2 3 / 3 1 / 3 2 / 3 3
  - 1 1 / 2 1 / 3 1 / 1 2 / 2 2 / 3 2 / 1 3 / 2 3 / 3 3
- Here, j is the inner loop and i is the outer loop
  - Inner loop (j) completed three times for each outer loop (i)

C and Python use row-major order



Iterations

```
for set in [1..3] do
    for rep in [1..3] do
        print set, rep
```

Iterations

- Iterations can be nested

```
for j in [1..3] do
    for i in [1..3] do
        print i, j
```

Unfold j

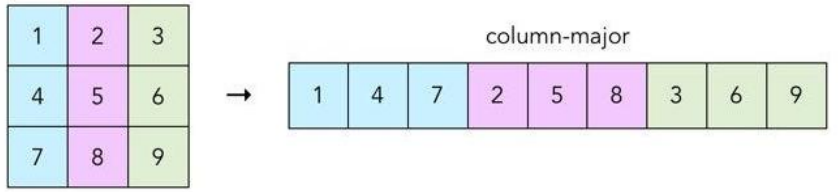
```
j = 1
for i in [1..3] do
    print i, j
j = 2
for i in [1..3] do
    print i, j
j = 3
for i in [1..3] do
    print i, j
```

$$A = \begin{bmatrix} 1,1 & 1,2 & 1,3 \\ 2,1 & 2,2 & 2,3 \\ 3,1 & 3,2 & 3,3 \end{bmatrix}$$

- This gives all combinations of i and j values (3 x 3 = 9 pairs), but what order?
  - 1 1 / 1 2 / 1 3 / 2 1 / 2 2 / 2 3 / 3 1 / 3 2 / 3 3
  - 1 1 / 2 1 / 3 1 / 1 2 / 2 2 / 3 2 / 1 3 / 2 3 / 3 3

- Here, i is the inner loop and j is the outer loop
  - Inner loop (i) completed three times for each outer loop (j)

Fortran uses
column-major order



Iterations

- Iterations can be nested even further

```
for i in [1..3] do
    for j in [1..3] do
        for k in [1..2] do
        print i, j, k
```

- This gives you all combinations of i, j and k values (3 x 3 x 2 = 18 tuples)
- List all tuples in correct order for these two programs.

```
for k in [1..2] do
    for i in [1..3] do
        for j in [1..3] do
        print i, j, k
```


That's all Folks!

Reminder: Next week Flipped Learning Lecture

Check your emails regularly for information about next week's flipped learning lecture