Python

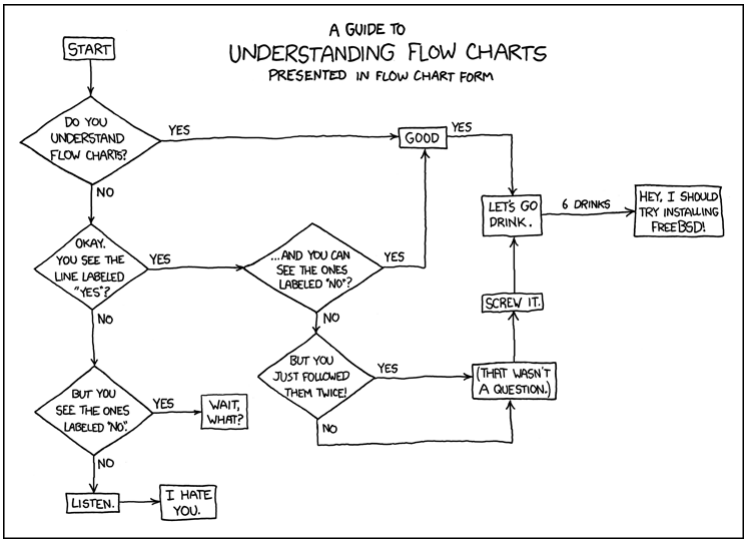
A quickstart into the key concepts of programming

Control structures

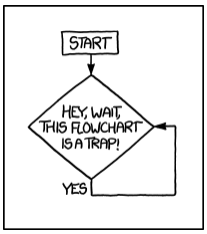
Control flow

Control flow

- Computer programs can't do that many things, they can:
  - Assign values to variables (memory locations).
  - Make decisions based on comparisons.
  - Repeat a sequence of instructions over and over.
  - Call subprograms.



https://xkcd.com/518/



https://xkcd.com/1195/

conditionals

control_flow_conditionals.ipynb

Conditional Statements: if

- Syntax:

```
if condition :
    indentedStatementBlock
```

- A group of individual statements, which make a single code block are called *suites*
- If the condition is true, then do the indented statements.
  If the condition is not true, then skip the indented statements.

```
if (a<100):
    print('a is less than 100')
```

- *File: if_1.py*

Conditional Statements: if else

- Syntax:

```
if condition :
    indentedStatementBlockForTrueCondition
else:
    indentedStatementBlockForFalseCondition
```

```
a = 88
if (a<10):
    print('a smaller than 10')
else:
    print('a larger than 10')
```

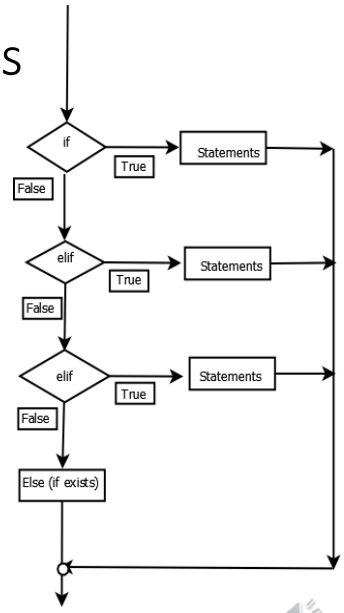
- *File: if_else_1.py*

Conditional Statements: chaining if's

- Combine several if statements into one statement using **elif**
- **else** block at the very end is not required

```
if x < 0:
     print("negative")
else:
      if x > 10:
          print("large")
      else:
          print("small")
```

```
if x < 0:
    print("negative")
elif x > 10:
    print("large")
else:
    print("small")
```



cs.uky.edu/~keen/115

loops

control_flow_loops.ipynb

Loops

- **for** loops are executed a fixed number of iterations. It is possible to exit early but this must be added to the code.
- **while** loops do not have a predetermined number of iterations. They terminate when some condition becomes False.

for loop

- **`for-in`**: loop over the sequence. for loops in Python require an array-like object (such as a list or a tuple) to iterate over.
- The general Python syntax:

```
for <item> in <iterable object>:
    blockToRepeat
else:  # optional
    blockElse
```

- *File: for_loop_1.py*

for loop

```
sum=0
numbers =[1,2,3,4]
for num in numbers:
    print(num)
    sum=sum+num  # check the indentation
avg=sum/len(numbers)
print ('Average:', avg)
```

For loop: else part

- use case: implement search loops, searching for an item meeting a particular condition, and need to perform additional processing or raise warning if no acceptable value is found
  - Loop should contain `break` statement.
  - The statements in the else block will be executed after all iterations are completed (normal termination).

```
numbers =[1,2,3,4]
for i in numbers:
    print(i)
    if i > 2.5:
       break
else: # Not executed as there is a break
    print("No Break")
```

while loop

- while loops use a condition to determine when to stop the loop.
- The general Python syntax:

```
while <boolean expression>:
    blockToRepeat
else:  # optional
    blockElse
```

- *File: while_loop_1.py*

Fine-Tuning Your Loops: break and continue

- The `break` statement breaks out of the loop entirely, any `else` clause will not be executed.
- The `continue` statement skips the remainder of the current loop, and goes to the next iteration
- *File: while_loop_break_continue.py*

iterator

- An iterator is a special object that gives values in succession, it implements the `next` protocol and raises `StopIteration` when exhausted.
- Iterator
  - `range()`
  - `enumerate()`
  - `zip()`
- The `for` loop automatically constructs an `iterator` from an `interator object`, then repeatedly calls `next` until a `StopIteration` is raised.

iterator

- The benefit of the iterator indirection is that *the full list is never explicitly created!*

```
N = 10 ** 12
for i in range(N):
    if i >= 10: break
    print(i, end=', ')
```

iter

- `iter` object is a container that gives you access to the `next` object for as long as it's valid

```
I = iter([2, 4, 6, 8, 10])
print(next(I))
2
print(next(I))
4
print(next(I))
6
```

for loop: range

- What if we need indexes?
  - Simulate a counter
  - use `range(len(our_list))` and lookup the index:

```
fruitslist = ['banana', 'apple',  'mango']
for i in range(len(fruitslist)):
    print(fruitslist[i])
```

- Source: http://treyhunner.com/2016/04/how-to-loop-with-indexes-in-python/

for loop: enumerate

- What if we need indexes?
- `enumerate` allows to loop over a list and retrieve both the index and the value of each item in the list:
- enumerate function returns an iterable where each element is a tuple that contains the index of the item and the original item value.

```
fruitslist = ['banana', 'apple',  'mango']
for num, color in enumerate(fruitslist):
    print(num, color)
```

- *File: for_loop_index*
- Source: http://treyhunner.com/2016/04/how-to-loop-with-indexes-in-python/

for loop: zip

- What if we need to loop over multiple things?
- `zip`: allows to loop over multiple lists at the same time
- This need is common enough that there's a special built-in function just for this.
- The zip function takes multiple lists and returns an iterable that provides a tuple of the corresponding elements of each list as we loop over it.

```
fruitslist = ['banana', 'apple',  'mango']
ratios = [0.2, 0.3, 0.1, 0.4]
for fruit, ratio in zip(fruitslist, ratios):
    print("{}% {}".format(ratio * 100, fruit))
```

- Source: http://treyhunner.com/2016/04/how-to-loop-with-indexes-in-python/

Comprehension

- Comprehension: powerful functionality within a single line of code; provides a compact way to create lists, dictionaries, sets
- `new_list = [`*`expression`* `for` *`member`* `in` *`iterable`*`]`
  - expression is the member itself, a call to a method, or any other valid expression that returns a value.
  - member is the object or value in the list or iterable.
  - iterable is a list, set, sequence, generator, or any other object that can return its elements one at a time.
- `squares = [i * i for i in range(10)]`

for loop: summary

- Loop over a single sequence with a regular `for-in`
- Loop over a list while keeping track of indexes with `enumerate`
- Loop over multiple lists at the same time with `zip`

for loop in other languages: Matlab

```
cellc = {'yellow', 'blue', 'black', 'white', 'cyan'};
lc = length(cellc);
for (i=1:lc)
    fprintf(" color = %s \n", cellc{i});
end
```

- A counter variable i is used
- The length of the cell array is calculated

Catching Exceptions

- Catching Exceptions: try and except
- Syntax

```
try:
    some statements here
except:
    exception handling
```

- The code within the try clause will be executed statement by statement. If an exception occurs, the rest of the try block will be skipped and the except clause will be executed.
- *File: try_except_1.py*

Accumulator pattern

- Common programming pattern:
  - walk through a sequence,
  - accumulate a value as you go,
  - at the end of the traversal, have single value accumulated
- The anatomy of the accumulation pattern includes:
  - initializing an “acccumulator” variable to an initial value (such as 0 if accumulating a sum)
  - iterating (e.g., traversing the items in a sequence)
  - updating the accumulator variable on each iteration (i.e., when processing each item in the sequence)

http://ice-web.cc.gatech.edu/ce21/1/static/audio/static/pip/Iteration/iteration.html

Accumulator pattern

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
accum = 0
for w in nums:
    accum = accum + w
print(accum)
```

File: accum_pattern_single.py

File: accum_pattern_single_text.py