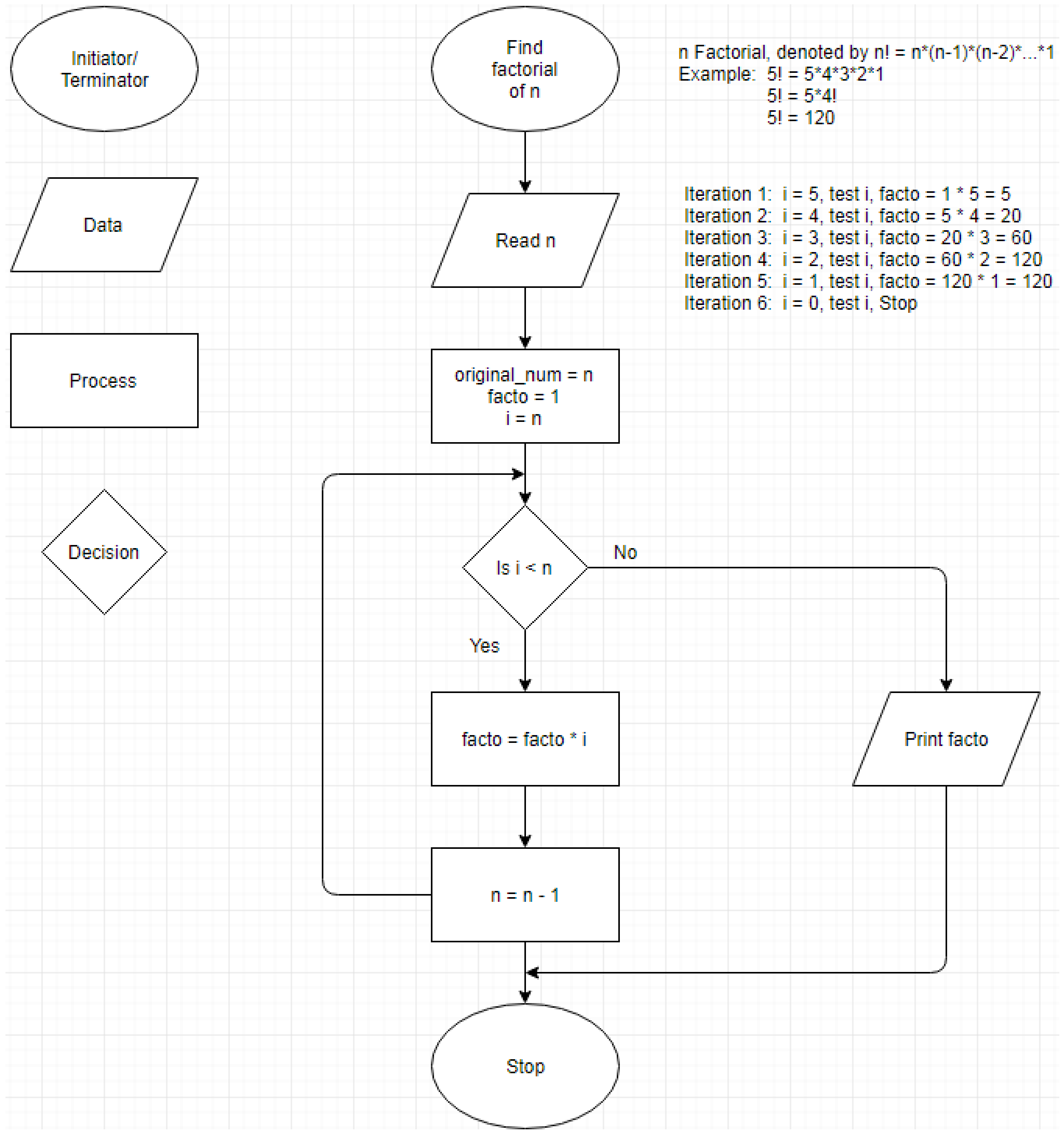
Flowcharts



n Factorial, denoted by n! = n*(n-1)*(n-2)*...*1
Example: 5! = 5*4*3*2*1
5! = 5*4!
5! = 120

Iteration 1: i = 5, test i, facto = 1 * 5 = 5
Iteration 2: i = 4, test i, facto = 5 * 4 = 20
Iteration 3: i = 3, test i, facto = 20 * 3 = 60
Iteration 4: i = 2, test i, facto = 60 * 2 = 120
Iteration 5: i = 1, test i, facto = 120 * 1 = 120
Iteration 6: i = 0, test i, Stop

Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about *coding style*. Most languages can be written (or more concise, *formatted*) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, **PEP 8** has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.

  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Wrap lines so that they don't exceed 79 characters.

  This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use `CamelCase` for classes and `lower_case_with_underscores`for functions and methods. Always use `self` as the name for the first method argument (see A First Look at Classes for more on classes and methods).
- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don't use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

https://docs.python.org/3/tutorial/controlflow.html#intermezzo-coding-style

Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

The Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Easter Egg

```
>>> import this
```

https://www.python.org/dev/peps/pep-0020/

Code examples

for_list_int.py

```
#!/usr/bin/env python3

x = 0
pos_ints = [1,2,3,4,5,6,7,8,9,10]
for pi in pos_ints:
    x += pi
print('The sum of these integers', pos_ints, 'is', x)
```

for_list_str.py

```
#!/usr/bin/env python3

words = ["I've", "seen", "things", "you", "people", "wouldn't", "believe."]
for w in words:
    print(w, len(w))
```

for_range.py

```
#!/usr/bin/env python3

n = int(input('Enter a number to compute factorial> '))
original_num = n
facto = 1

for i in range(n, 0, -1):
    facto = facto * i
    n -= 1
print(str(original_num) + '! =', facto)
```

for_primes.py

```
#!/usr/bin/env python3

x = int(input('Please enter the beginning of integer range> '))
y = int(input('Please enter the end of integer range> '))
for m in range(x, y):
    '''
    Notice how this for statement has an else branch.
    What's up with that?
    '''
    for n in range(2, m):
        if m % n == 0:
            print(m, 'equals', n, '*', m//n)
            break
    else:
        # the Loop found a prime number
        print(m, 'is a prime number')
```

if_int.py

```
#!/usr/bin/env python3.6

'''
Notice the cast to int on the line below
'''
x = int(input('Please enter an integer: '))

if x < 0:
    print(x, "Hey, that's a negative number!")
elif x == 0:
    print('You typed in 0 (zero)')
elif x == 1:
    print('You typed in 1 (how boring)')
else:
    print(x, "You life on the ragged edge of life!!!")
```

if_product.py

```
#!/usr/bin/env python3.6

'''
Doing math operations using the inputs
'''

'''
Notice the cast to int on the input
'''
x = int(input('Please enter an integer> '))
y = int(input('Please enter another integer> '))
the_product = x*y
'''
You will often use the if, elif, else structure.
'''
if the_product > 10000:
    print(x,'*', y, '=', x*y, "Now that's a big number!")
elif the_product == 0:
    '''
    Notice how to continue a long string on the next line.
    80 char width is a good Python practice.
    '''
    print(x,'*', y, '=', x*y, "Zero is the only int which is neither positive"
          "or negative")
elif the_product < 0:
    print(the_product, "I guess it's OK to have negative numbers")
else:
    print(x,'*', y, '=', x*y, "That number is not too big")
```

while_true.py

```
#!/usr/bin/env python3

'''
Notice that this is an infinite while loop.
The loop will exit because of the "break"
statement
'''
while True:
    n = input("Please enter the super secret word> ")
    if n.strip() == 'supercalifragislistic':
        break
    else:
        print("Nope.  Try again.")
```

while_expression.py

```
#!/usr/bin/env python3

'''
Notice that this while loop stops when a
certain condition is met.  In this case
we do not use the "break" statement
'''
n = input("Please enter the super secret word> ")

while n != 'supercalifragilistic':
    print("Nope.  Try again.")
    n = input("Please enter the super secret word> ")
print("You entered", n, "Good job!")
```

while_ints.py

```
#!/usr/bin/env python3

x = int(input("Please enter the first number> "))
y = int(input("Please enter the second number> "))

'''
Here is another example of a while loop that exits
when the "expression" becomes false
'''
while x < y:
    if x % 2 != 0:
        x += 1
        continue
    print(x, "is an even number")
    x += 1
```

while_fibonacci.py

```
#!/usr/bin/env python3

'''
In this example we compute a Fibonacci series
'''

# Notice how we can initialize two variables on one line
a, b = 0, 1

n = int(input("Input the end of the Fibonacci series> "))
while b < n:
    print(b)
    a, b = b, a + b
```

continue.py

```
#!/usr/bin/env python3

for letter in "noodle poodle bottled paddled muddled duddled fuddled wuddled":
    if letter == "d":
        continue
    print(letter, end="")
print()
```

mortgage1.py

```
#!/usr/bin/env python3.6

principal = 500000
payment = 2684.11
rate = 0.05
total_paid = 0
month = 0

print('{:>5s} {:>10s} {:>10s} {:>10s}'.format('Month', 'Interest', 'Principal',
                                              'Remaining'))
while principal > 0:
    month += 1
    interest = principal*(rate/12)
    principal = principal + interest - payment
    total_paid += payment
    print('{:>5d} {:>10.2f} {:>10.2f} {:>10.2f}'.format(month, interest,
                                                        payment - interest,
                                                        principal))
print('Total paid: %10.2f' % (total_paid))
```

mortgage2.py

```
#!/usr/bin/env python3.6

principal = 500000
payment = 2684.11
rate = 0.05
total_paid = 0
month = 0

# Extra payment info
extra_payment = 1000
extra_payment_start_month = 1
extra_payment_end_month = 60

print('{:>5s} {:>10s} {:>10s} {:>10s} {:>10s}'.format('Month', 'Total',
                                                      'Interest', 'Principal',
                                                      'Remaining'))
print('{:>5s} {:>10s} {:>10s} {:>10s} {:>10s}'.format('', 'Payment', 'Paid',
                                                      'Paid', 'Principal'))

while principal > 0:
    month += 1
    if month >= extra_payment_start_month and month <= extra_payment_end_month:
        total_payment = payment + extra_payment
    else:
        total_payment = payment
    interest = principal*(rate/12)
    principal = principal + interest - total_payment
    total_paid += total_payment
    print('{:>5d} {:>10.2f} {:>10.2f} {:>10.2f} {:>10.2f}'.format(month,
                                                                  total_payment,
                                                                  interest,
                                                                  total_payment -
                                                                  interest,
                                                                  principal))
print('Total paid: %10.2f' % (total_paid))
```

http://www.dabeaz.com/

Further examples of break, continue, and pass

pass is a no-operation command. It is parsed, does nothing, and continues on to the next operation

```
>>> while True:
...     pass
...
^CTraceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

continue returns the control to the beginning of the while loop

```
>>> while True:
...     continue
...
^CTraceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

break, like in C, breaks out of the smallest enclosing **for** or **while** loop

```
>>> while True:
...     break
...
```