Module 3: While Loops

Exercise

If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-03-start.ipynb
```

Discuss the previous module with your neighbour.

- What are `bool` values, and what can we do with them?
- How exactly do you write an `if` statement with many branches?

One basic thing we often want to do with computers is to do something repeatedly.

For example, to count down from 5 to 1, I could write:

```
print(5)
print(4)
print(3)
print(2)
print(1)
print("Blastoff!")
```

But this looks like work, and I'm lazy.

If I wanted to do the same thing starting at 100, it would be a lot of work.

There must be a better way, and there is: we can use a **loop**.

`while` loops

The simplest way to repeat is to something, over and over again, until the task is complete. Examples:

- To wash the dishes:
  While there are dishes left, wash a dish.
- To play chess:
  While you have not yet won or lost, make a move.
- To count down from *n* to zero:
  While *n* is not zero, say *n*, then make *n* smaller.

```python
def countdown(n: int) -> None:
    """Count down from n to zero."""
    while n != 0:
        print(n)
        n = n - 1

    print("Blastoff!")
```

Syntax of `while`

The syntax of `while` is similar to the syntax of `if`.

We write `while`, then a Boolean expression, then a colon, followed by a block of code.

The difference is in the interpretation; instead of possibly running the code once, it runs it repeatedly, zero or more times, as long as the Boolean expression is `True`.

```
total = 0
n = 5
while n > 0:
    total = total + n
    n = n - 1
```

```
x = 1
while x < 1000:
    print(x)
    x = x * 2
```

Now that we have `while` loops, **state diagrams** become very important.

Ex. Use a state diagram to work through what each of these snippets does.

Working with `while`

Let's turn one of these into a function →

Exercise

Following this pattern, write a function to return the sum of the squares, e.g.
`sum_squares(4)` ⇒ `4*4 + 3*3 + 2*2 + 1*1` ⇒ `30`

The **factorial function**, written $n!$, is the product of the positive integers up to n.
For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

Exercise

Write a function `factorial(n)` that calculates $n!$.

```
def sum_to(n: int) -> int:
    """Return the sum 1 + 2 + ... n.
    Requires: n >= 0."""
    total = 0
    while n > 0:
        total = total + n
        n = n - 1

    return total

check.expect("s3", sum_to(3), 3+2+1)
check.expect("s5", sum_to(5),
             5+4+3+2+1)
```

Exercise

Write a function `sum_between(lo, hi)` that returns the sum of integers from `lo` to `hi`.
For example, `sum_between(12, 15)` ⇒ `12 + 13 + 14 + 15` ⇒ `54`

Not just counting: dividing out 2

So far, we have always been just counting down (or up). We could always tell in advance how many times the loop would execute. This isn't always the case.

I ask: "how many times can I divide a positive number by 2 until I get below 2?"

For example, $12 = 2 \times 6$, $6 = 2 \times 3$, $3 = 2 \times 1.5$. I can divide 12 three times. And $27 = 2 \times 13.5$, $13.5 = 2 \times 6.75$, $6.75 = 2 \times 3.375$, $3.375 = 2 \times 1.6875$. I can divide 27 four times.

Let's write a function that does this.

What do we need to keep track of? At least:

1. how big is our number still ($12 \rightarrow 6 \rightarrow 3 \rightarrow 1.5$)
2. how many times we have divided so far ($0 \rightarrow 1 \rightarrow 2 \rightarrow 3$)

```
def count_twos(n: float) -> int:
    """Determine how many times n can be
    divided by 2 until we get below 2."""
    count = 0
    while n >= 2:
        count = count + 1
        n = n / 2

    return count

check.expect("C12", count_twos(12.0), 3)
check.expect("C27", count_twos(27.0), 4)
```

Example: Collatz Sequence

Starting from any positive integer `n`, I form a sequence of integers using this simple rule:

- if `n` is even, the next value in the sequence is `n // 2`
- if `n` is odd, the next value in the sequence is `3 * n + 1`

For example, starting at 3:

- 3 is odd, so the next value is 10
- 10 is even, so the next value is 5
- 5 is odd, so the next value is 16
- 16 even, so the next value is 8
- 8 even, so the next value is 4
- 4 even, so the next value is 2
- 2 even, so the next value is 1
- 1 odd, so the next value is 4...

It **seems** that from any starting point, the sequence always eventually reaches 1.

Exercise

Write a function `collatz_len(n: int) -> int` that determines how many steps the Collatz sequence takes to get from `n` to 1.

We need to keep track of:

- `n`, which will change
- how many steps we've taken.

Example: Longest Collatz Sequence

Suppose we want to find the length of the longest Collatz sequence that starts below some integer `top`. To understand the problem better, let's try:

```
collatz_len(1) ⇒ 0
collatz_len(2) ⇒ 1
collatz_len(3) ⇒ 7
collatz_len(4) ⇒ 2
collatz_len(5) ⇒ 5
```

We need to keep track of

1. a counter of where we start,
2. the longest length we've seen so far.

Exercise

Write a function `longest_collatz(top)` that returns the length of the longest Collatz sequence starting between 1 and `top`.

```
longest_collatz(5) ⇒ 7
```

Exercise

Rework this function to write a function `longest_start(top)` that instead it returns the **starting value** of the longest sequence.

```
longest_start(5) ⇒ 3
```

In this variant we also need to store **what value** we saw this longest sequence from.

Example: Factorizing

Every positive integer can be written as a product of prime factors.

For example:

- $12 = 2 \cdot 2 \cdot 3$
- $60 = 2 \cdot 2 \cdot 3 \cdot 5$
- $77 = 7 \cdot 11$

It often helps to draw a "tree" to determine this. We keep dividing out the smallest number possible, until we can't divide it out any more. Then try the next smallest number.

We need to keep track of: 1 what is left, and 2 what we're trying to divide by.

Exercise

Write a function `factorize(n: int) -> int`. It shall print the prime factors of *n* in increasing order, and return an `int` indicating how many there are.
For example, `factorize(60)` should print `2`, `2`, `3`, `5`, and return 4.

Example: calculating square roots

To estimate the square root of a non-negative number n, we seek g such that $g^2 = n$.

We're going to start with a guess, then make it better, until it's "good enough".

We "want" $g^2 = n$. Rewrite this as $g = \dfrac{n}{g}$.

If g is "too small", then $\frac{n}{g}$ is "too big", and vice-versa.

The answer is guaranteed to be between g and $\frac{n}{g}$. Any number between them is a better guess! Pick any number between them... how about right in the middle (the average).

So a better guess is $g' = \frac{g+\frac{n}{g}}{2}$. Repeatedly improve the guess until g^2 is very close to n.

```python
def sqrt(n: float) -> float:
    g = 1.0 # initial guess; it may be bad, but it doesn't matter.
    ## Don't start at the int 1; we promised to return a float!
    while abs(g**2 - n) > 0.0001:
        g = (g + n/g) / 2
    return g
```

It turns out that the trigonometry function cos can be calculated using:

$$\cos x = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \dots$$

(Note that 0! is 1. Often, including here, $0^0 = 1$. Calculate $n!$ using `math.factorial`.)

We want to stop when the next term is close to zero. We need to keep track of:

1. the total,
2. a counter,
3. the sign (+ or –)

Exercise

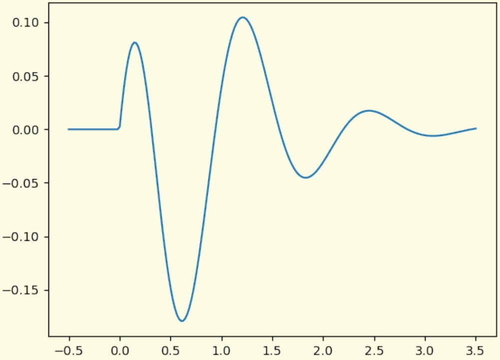
Write a function `cos(x)` that uses a **`while`** loop to calculate this value, stopping when the next term is smaller than 0.0001.

Do not use any `math` functions except `math.factorial`.

We can now write functions.
Next we are going to consider how to write **functions that look at functions.**

Here's a plot of a function. I might ask:

1. For what values of x is this function zero?
2. What is the area under the curve?
3. ...



We don't want to re-write our code for each function.

We want to write code that can answer such questions for any function. We only need to write such code once.

A Function is a value

We are used to values of type `int`, `float`, `str`, and `bool`.

A function is also a value. We can assign it to a variable:

```
q = abs
q(-3) ⇒ 3
q(4) ⇒ 4
help(q)
```

This `q` is just as good as `abs`; in fact it's exactly the same thing.

xercis

Consider carefully: what is the difference between `p = abs(-3)` and `q = abs` ?

- The value of `p` comes from **calling** the function `abs` with argument `-3`. The function returns the value `3`, so `p` takes the value `3`, which is an `int`.
- Since we do not have brackets `()` after `abs`, **we are not calling this function**. The value of `q` is `abs` itself.

A Function as a parameter

We can assign a function to a variable; we can also use a function as an argument to a function.

To annotate a parameter that is a function, we will write `callable`.

```
def call_n_times(n: int, f: callable) -> float:
    """Countdown from n to 0, print f for each value,
    and return their total.
    """
    total = 0.0
    while n > 0:
        print("f(", n, ") =>", f(n))
        total = total + f(n)
        n = n - 1

    return total
```

Note that `f` is a parameter. But it's also a function, and to **call** it, we need to write it with **brackets** and **argument(s)**.

Example: first negative

Exercise

Write a function `first_negative(f: callable) -> int`. It takes a `callable`, and returns the smallest natural number for which `f` returns a negative number.

To have an example, we need to define a function to call `first_negative` with.

```
def trajectory(x: float) -> float:
    """Return the y coordinate on a particular trajectory at x."""
    return - (x + 3.2) * (x - 4.6)
```

`trajectory(0) > 0`, `trajectory(1) > 0`, ..., `trajectory(4) > 0`, but `trajectory(5) < 0`.

So `first_negative(trajectory)` should return `5`.

And consider `math.cos`. `math.cos(0) > 0` `math.cos(1) > 0`, but `math.cos(2) < 0`.

So `first_negative(math.cos)` should return `2`.

! Note: `first_negative` will not directly call `trajectory` or `math.cos`. It will call only `f`.

Examining a function

Let's specify **how many** times to call a function, **evenly spaced** in some interval.

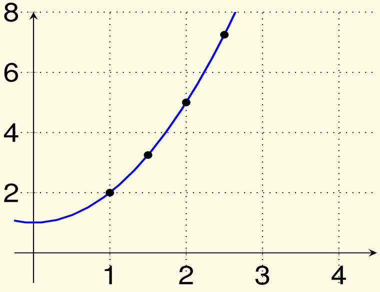
For an example, let's define a function:

```
def parabola(x: float) -> float:
    return x**2 + 1
```

Imagine we call it 4 times, with the first at $x = 1.0$, and the last point is **just before** $x = 3.0$.

So `print_interval(parabola, 1.0, 3.0, 4)` should print:

```
1.0 -> 2.0
1.5 -> 3.25
2.0 -> 5.0
2.5 -> 7.25
```



A plot of $f(x) = x^2 + 1$

Exercise

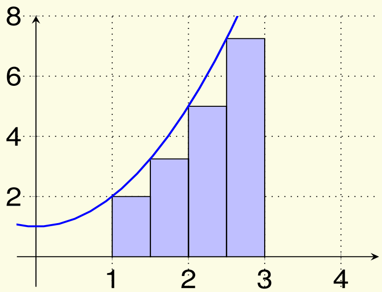
Write the function `print_interval(f, x0, x1, count)` that makes `count` calls to the function `f`, evenly spaced starting and `x0` and ending just before `x1`.

Approximating the area under a curve

The area of a rectangle is $b \times h$ where b and h are the base and height.
We can estimate the area of any weird shape by adding up a lot of little rectangles.

The area of $f(x) = x^2 + 1$, using 4 bins between 1.0 and 3.0, is approximately:

$$2.0 \times 0.5 + 3.25 \times 0.5 + 5 \times 0.5 + 7.25 \times 0.5 = 8.75$$



A plot of $f(x) = x^2 + 1$

Exercise

Write a function `approx_area(f: callable, x0: float, x1: float, nbins: int) -> float`.
The function returns an approximation of the area of between `f` and the x-axis, between `x0` and `x1`, using `nbins` bins. For example:

```
check.within("parabola", approx_area(parabola, 1.0, 3.0, 4), 8.75, 0.0001)
check.within("sin", approx_area(math.sin, 0.0, math.pi, 1000), 2.0, 0.0001)
```

Module summary

- Use `while` loops with a counter to write code where we can directly see how many times the loop will be executed.
- Use `while` loops to write code where the end condition cannot be directly identified, but depends on the calculation.
- Write functions that have a function as a parameter.

Before we begin the next module:

- Read and complete the exercises in module 3 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 3 Review Quiz, due on Monday.