Parameters

Quest

Now that you have seen the basic anatomy of a function, we can dive a little deeper into each part. First up is parameters. We told you before how parameters are variables that need to be assigned a value before the function can run. Arguments are the information coming from the function call, and parameters are the variables defined in the function definition. The purpose of this section is to further introduce you to the world of parameters so that you can properly use them to create your own functions.

Choosing Parameters

When you're getting ready to write a function, you can take a moment to think about what your function does, what information it needs to do that, and how much of that information it already has.

For example, I want to write a function called `isEven` that can tell you whether an integer is even or odd. Ask yourself, *what do I need to write this function?*

Things you'll need:

1. A number
2. A way to tell if that number is even
3. A way to tell if that number is odd
4. A way to return if that number is even or odd

Items 2, 3, and 4 all use built-in features such as `return` and Python's built-in math operators. But what about item 1? We don't yet know which number needs to be evaluated as even or odd. The whole point is that this function will work on *any* integer. Perhaps we could use a *placeholder variable* for the integer we want to evaluate and then *pass* in whichever integer we like to replace that placeholder. *Ding Ding Ding!* This is a perfect opportunity to use a parameter. As long as our code works generally for all integers, this function will work on any integer we throw at it.

So, what do you think our function definition will look like? We need a name for the function and a name for the parameter that will hold our integer. Let's call that parameter `number`.

```
def isEven(number):
```

Now onto the body! To determine whether any integer is even or odd, we can employ the modulus (`%`) operator to get the job done. If a number divided by 2 has no remainder, then that number is even. So, if the name `number` refers to the integer we are evaluating, we don't need to know the value of `number` to determine if it is even! Once we know our answer, we can give an answer back to the function call using `return`.

```
def isEven(number):
'''
This function evaluates whether a given integer is even or odd
params:
    number : The integer to be evaluated
return:
    True if number is even, false otherwise
'''
if number % 2 == 0: # Number is even
    return True
else:               # Number is odd
    return False
```

Now we can write a full program that uses `isEven` as a helper function! This program will take in a number from the user and then print whether or not that number is even:

```
def isEven(number):
    '''
    This function evaluates whether a given integer is even or odd
    params:
        number : The integer to be evaluated
    return:
    True if number is even, false otherwise
    '''
    if number % 2 == 0: # Number is even
        return True
    else:               # Number is odd
        return False


def main():
    user_input = int(input("Please enter your favorite number: "))
    if isEven(user_input):
        print(user_input, "is even!")
    else:
        print(user_input, "is odd!")

if __name__ == '__main__':
    main()
```

```
Please enter your favorite number: 23
23 is odd!
```

▶ Run >_ Hide

Lifecycle of an Argument

To better illustrate what happens when we use arguments and parameters, we are going to walk through the above program and watch how information gets passed to and from each function!

Multiple Parameters

Let's say you want to create a function called `repeat` that takes in a string `s` and a number `n` and repeats the string `s` times. For example, calling `repeat('ab', 3)` would return `'ababab'`.

```
def repeat(s, n):
    '''
    repeats the input string s, n times
    params:
        s (str): string to repeat
        n (int): number of times to repeat it
    return: repeated string
    '''
    repeat_str = ''
    for i in range(n):
        repeat_str += s

    return repeat_str


def main():
    print(repeat('ab', 3))


if __name__ == '__main__':
    main()
```

```
ababab
```

▶ Run >_ Hide

There are several things to note from this example. First, as we can see, functions can have multiple parameters. When we define parameters, we often make an assumption about the type of information going into that parameter. If we created a parameter to store someone's name, we'd probably expect that to be a `string` instead of, say, a `float`. Even though you can theoretically pass any type of data in for each parameter. Python does not check the type of parameter that you pass into a function unless you ask it to. This can be both useful and harmful, depending on the type of function you are trying to write. This is useful if you want your function to work for multiple types, but it can lead to 🔴 errors if a user calls your function with arguments that have a type your function was not designed for:

```
def repeat(s, n):
    '''
    repeats the input string s, n times
    params:
        s (str): string to repeat
        n (int): number of times to repeat it
    return: repeated string
    '''
    repeat_str = ''
    for i in range(n):
        repeat_str += s

    return repeat_str


def main():
    print(repeat(4, 3))


if __name__ == '__main__':
    main()
```

```
  CodeRunner(
File "/lib/python3.9/site-packages/_pyodide/_base.py", line 296, in run
  coroutine = eval(self.code, globals, locals)
File "<exec>", line 3, in <module>
File "unthrow.pyx", line 53, in unthrow.Resumer.run_once
File "<exec>", line 26, in mainApp
File "<exec>", line 22, in main
File "<exec>", line 16, in repeat
TypeError: can only concatenate str (not "int") to str
```

▶ Run >_ Hide

Another important thing to note is that the order of arguments matters. In this case, our function expects the `s` parameter to be first. If we attempted to swap them in our function call, we would get an error 🔴 because our function was written such that we expected the first argument to be a `string`:

```
def repeat(s, n):
```

```
    '''
    repeats the input string s, n times
    params:
        s (str): string to repeat
        n (int): number of times to repeat it
    return: repeated string
    '''
    repeat_str = ''
    for i in range(n):
        repeat_str += s

    return repeat_str


def main():
    print(repeat(3, 'abab'))


if __name__ == '__main__':
    main()
```

```
  CodeRunner(
File "/lib/python3.9/site-packages/_pyodide/_base.py", line 296, in run
  coroutine = eval(self.code, globals, locals)
File "<exec>", line 3, in <module>
File "unthrow.pyx", line 53, in unthrow.Resumer.run_once
File "<exec>", line 26, in mainApp
File "<exec>", line 22, in main
File "<exec>", line 15, in repeat
TypeError: 'str' object cannot be interpreted as an integer
```

▶ Run >_ Hide

We can also add **default values** for each parameter. A default value gives the parameter a sort of backup option to use in case no argument is given in the function call. If an argument is given, that argument overrides the default value and is used instead. If a parameter doesn't have a default value, it *must* be given an argument from the function call. We can see the syntax for default values below:

```
def repeat(s = 'cd', n = 4):
    '''
    repeats the input string s, n times
    params:
        s (str): string to repeat
        n (int): number of times to repeat it
    return: repeated string
    '''
    repeat_str = ''
    for i in range(n):
        repeat_str += s

    return repeat_str


def main():
    print(repeat('ab', 3))
    print(repeat())
    print(repeat('bc'))


if __name__ == '__main__':
    main()
```

```
ababab
cdcdcdcd
bcbcbcbc
```

▶ Run >_ Hide

In the last example, we see that even though only one argument was given, Python still correctly matched it to the first parameter. As we mentioned earlier, order matters. That being said, if we only wanted to change the default value for `n` there is a way to do that. We just have to tell Python which parameter we are setting in the function call:

```
def repeat(s = 'cd', n = 4):
    '''
    repeats the input string s, n times
    params:
        s (str): string to repeat
        n (int): number of times to repeat it
    return: repeated string
    '''
    repeat_str = ''
    for i in range(n):
        repeat_str += s

    return repeat_str


def main():
    print(repeat(n = 2))


if __name__ == '__main__':
    main()
```

```
cdcd
```

▶ Run >_ Hide

We've spent this entire section telling you that order matters when *calling* functions. Well, it turns out

it matters when declaring them as well. All default values must come after normal parameters. Otherwise, you will get an error 🔴 :

```python
def repeat(s = 'cd', n):    # this will cause an error
    '''
    repeats the input string s, n times
    params:
        s (str): string to repeat
        n (int): number of times to repeat it
    return: repeated string
    '''
    repeat_str = ''
    for i in range(n):
        repeat_str += s

    return repeat_str


def main():
    print(repeat('heyo', 3))


if __name__ == '__main__':
    main()
```

```
Traceback (most recent call last):
  File "/lib/python3.9/site-packages/_pyodide/_base.py", line 415, in eval_code
    CodeRunner(
  File "/lib/python3.9/site-packages/_pyodide/_base.py", line 237, in __init__
    self.ast = next(self._gen)
  File "/lib/python3.9/site-packages/_pyodide/_base.py", line 141, in _parse_and_compi
    mod = compile(source, filename, mode, flags | ast.PyCF_ONLY_AST)
  File "<exec>", line 6
    def repeat(s = 'cd', n):    # this will cause an error
```

▶ Run >_ Hide