# Python String Interpolation

- Also known as variable interpolation
- What is it?

# String Interpolation: What is it?

- Is the process of evaluating a string literal containing one or more placeholders
- Yields a result in which the placeholders are replaced with their corresponding values.

# String Interpolation Example

```
def main():
    # in Python 2
    bananas = 3
    print "I have %d bananas" % bananas
    print "I have %(bananas)d bananas" % locals()
```

# String Interpolation vs Concatenation

```
#Concatenation is no bueno
def main():
    bananas = 3
    print "I have " + bananas + " bananas"
```

- Allows for easier and more intuitive string formatting and content-specification when compared to string concatenation

# Goals

- Eliminate need to pass variables manually
- Eliminate repetition of identifiers and redundant parentheses

# Goals

- Remove awkward syntax
- Eliminate mismatch errors
- Avoid need for locals() and globals() usage, parsing the given string and passing in named parameters automatically

# Limitations

- Backwards compatibility
- Python specifies both single and double quotes to enclose strings, so it is not reasonable to choose one of them now to enable interpolation.
- ‘ (accent) a shortcut for repr()

# Remaining Options

- Operator for printf style string formatting via %
- Class, such as string.Template()
- Method or Function, such as str.format
- New Syntax

# Implementation: Printf formatting via operator

`Pros:`

- Simple
- Similar to string interpolation methods used in other programming languages.

# Implementation: Printf formatting via operator

```
>>> params = {'user': 'nobody', 'id': 9, 'hostname':
    'darkstar'}

>>>'Hello, user: %(user)s, id: %(id)s, on host:
     %(hostname)s' % params

 Hello, user: nobody, id: 9, on host: darkstar
    #Output
```

# Implementation: Printf formatting via operator

`Cons:`

- Can only take in one argument besides original string.
- Multiple parameters require a passed in dictionary or tuple.
- Easier to make syntax errors

# Implementation: stringTemplate Class

```
>>> params = {'user': 'nobody', 'id': 9, 'hostname':
    'darkstar'}

Template('Hello, user: $user, id: ${id}, on host:
$hostname').substitute(params)
```

- Uses safe-substitution, which silently ignores malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

# Implementation: str.format()

```
# when using keyword args, var name shortening
# sometimes needed to fit :/
>>> 'Hello, user: {user}, id: {id}, on host: {host}'
.format(user=user, id=id, host=hostname)
'Hello, user: nobody, id: 9, on host: darkstar'
```

- Extremely Verbose, not very convenient

# New Syntax(Last Resort)

- New syntax is generally avoided, in the interest of backwards compatibility.
- Build off of str.format()

# New Syntax Proposal

```
>>> location = 'World'
>>> f'Hello, {location} !'          # new prefix: f''
'Hello, World !'                    # interpolated result
```

# Safety

- Only string literals can be considered for format-strings
- Neither `locals()` nor `globals()` are used during transformation
- Recursive interpolation not supported

# Precautions

- Mistakes or malicious code can be obscured inside a string
- Highly recommended to avoid constructs in format-strings

# Backwards Compatibility

```
# uses existing syntax and avoids historical features
# format strings were designed to be backwards compat
```

# Internalization

```
# deemed too difficult to implement
```

- Use-cases differ
- Compile vs. run-time tasks
- Interpolation syntax requirement
- Does not necessarily match intended audience
- Security policy risks

# Additional Rejected Ideas

- Limiting syntax to str.format()
- Additional / Custom String-Prefixes
- Automated Escaping of Input-Variables
- Environment Access and Command Substitution