Nuitka: All Python built-ins optimized for Nuitka (sub-org: Nuitka)

Google Summer of Code 2019 Proposal

1. About Me

1.1 Basic information

Full name	Batakrishna Sahu
University	Sambalpur University Institute of Information Technology
Program	Bachelor Of Technology in Computer Science and Engineering
Expected graduation	May 2020
E-mail Address	bablusahoo16@gmail.com
GitHub	https://github.com/bksahu
Hangouts	bablusahoo16@gmail.com
Twitter	https://twitter.com/bksahu_
Gitter	bksahu
Phone	
Time Zone	Indian Standard Time (UTC +5:30)

1.2 Background and programming experience

I am a third-year undergraduate student at Sambalpur University Institute of Information Technology majoring in Computer Science and Engineering. I started programming in Python one year ago and ever since been in love with it. Apart from Python and I also have a knowledge of Javascript, C, and Java. I use `git` for all my Git-related needs and Eclipse for Python development. I have also completed the #100DaysOfMLCode challenge.

1.3 Code Contribution

I have been actively contributing to open source projects in Python and JavaScript and also been contributing to Nuitka for a while now. I started contributing to **Nuitka** by doing small doc-fixes and writing test cases. It was when working on optimizing “any” built-in which helped me a lot to get familiar with the codebase and debugging. It has also been a great experience getting involved with the “Chief Creative Officer” and project founder Kay Hayen which took my understanding of **Nuitka** to the next level.

Patchs Submitted:

PR	Description	Status
#246	Code: Adding support for "any" built-in	open
#333	Tests: Added support for "only" mode in Test runners	merged
#290	Tests: Cover more Trick Assignments tests	merged
#247	Doc: Fixed API doc logo size	merged
#250	Doc: Fixed the Isort project link in README.rst	merged
#252	Doc: Fix developer manual typo	merged
#273	Doc: Improve slots example	merged
#336	Quality: Fixed the TypeError in autoformat	merged

Issues Reported:

PR	Description	Status
#334	Autoformat of Nuitka source code gives TypeError and UnicodeDecodeError	open
#320	Running a single test	fixed

2. Project Information

2.1 Sub-org Introduction

Nuitka is a Python compiler that can compile every construct that CPython offers. It translates the Python code into a C program that then is linked against `libpython` to execute in the same way as CPython does, in a very compatible way.

2.2 Project Abstract

In **Nuitka,** there is a specialized node for every built-in that is to be optimized. Every time a built-in is called in the code, this specialized node tries to compute the expression during the compile time resulting in early exits. But there are many built-ins yet to be optimized which can have a high-performance impact in some cases. The proposed project aims to identify all the missing Python 2.7 to 3.7 (and 3.8 eventually) built-ins in **Nuitka** and to optimize them.

2.3 Project details

2.3.1 Current state and implementation

Currently, there are a total of 69 built-ins in Python 3 and 76 built-in in Python 2 out of which 27 built-ins are yet to be optimized in

Nuitka. A status list of those missing built-ins can found below in the tentative timeline section.

Every specialized built-in node has an entry in the dictionary `_dispatch_dict` that corresponds to an extractor code which tries to optimize during the compile time. For the purpose of illustration, we will use the “any” node here:

```
def any_extractor(node):
    return BuiltinParameterSpecs.extractBuiltinArgs(
        node=node,
        builtin_class=ExpressionBuiltinAny,
        builtin_spec=BuiltinParameterSpecs.builtin_any_spec,
    )

_dispatch_dict = {
    "node": any_extractor,
    ...
```

nuitka.optimizations.OptimizeBuiltinCalls

As we can see in the `any_extractor` the `builtin_class` points to the `ExpressionBuiltinNode` where the Python side of the optimization takes place. These specialized nodes are the children of `ExpressionBases` from where they inherit methods like `computeExpression`.

```
class ExpressionBuiltinAny(ExpressionBuiltinSingleArgBase):
      kind = "EXPRESSION_BUILTIN_ANY"
      builtin_spec = BuiltinParameterSpecs.builtin_Node_spec

      def computeExpression(self, trace_collection):
            return self.getValue().computeExpressionAny(
                  Node_node=self, trace_collection=trace_collection
            )
```

nuitka.nodes.BuiltinNodeAny

There is a method called `computeExpressionAny` in the `ExpressionBases` module that tries to predict the value and does constant folding optimization. This method returns a tuple of (node, tags, description) that gives information about which line is being optimized in the code.

```
def computeExpressionAny(self, node_node, trace_collection):
                 value = any_node.getValue()
                 shape = value.getTypeShape()

                 ...



                 return (
                     result,
                     "new_constant",
                     "Predicted 'Any' result from value shape.",
                 )
        # in unknown cases, allow for exceptions
        # and unknown code to execute (control flow and values
        # escaped)
        self.onContentEscapes(trace_collection)
        trace_collection.onControlFlowEscape(self)
        trace_collection.onExceptionRaiseExit(BaseException)

        # if there is no optimization
        return any_node, None, None
```

nuitka.nodes.ExpressionBases

There is a code generation which points to a function that emits C code with an entry in the `setExpressionDispatchDict` dictionary. The C code has to have a similar implementation as of CPython.

2.3.2 Optimization

This is the most exciting part about working on **Nuitka**. It plays a crucial role in improving the performance of Python. **Built-in call prediction** is the most often used optimization technique while optimizing the built-ins. It can be thought as of constant folding but for built-in expressions rather than constants. It is often possible in case of built-in calls like `len`, `any` and `range` to predict the result at compile time rather than at runtime. For example:

```
any([None, None, None]) # predictable
any([0]*2000) # predictable
```

However, the cases where high computation is involved should be avoided. For example:

```
any([0]*100000) # predictable but high computation require
```

2.3.3 My Approach

I will strive to follow a Test Driven Development (TDD) pattern. My general workflow while optimizing any new built-in will be like the following:

1. Research about the built-in and specifically try to find the answers to these questions:
    - What are the parameters and return type shapes?
    - Does the given built-in have or use slots?
    - What are the side effects?

    I will also go through original CPython implementation and try to find all the possible optimizations.
2. Once I have listed all the possible optimizations then I will move on to write relevant tests.
3. Write the optimization code.
4. Add the documentation.

3. Tentative Timetable

This is merely a modest sketch. I have tried to be as lenient as possible in assigning the weekly tasks. Every week I will try to optimize 2 to 3 built-ins in the order listed below. Although work on many built-ins should not depend on external needs, it could happen that some built-in might have to be postponed because e.g. relevant documentation of Nuitka internals is not immediately available, or that hard to identify bugs cause delay. In these instances, other built-ins might be started sooner, and problematic ones might finish later, while easier ones finish early. I am planning to dedicate at least 40 hours per week (Mon-Sat). Though if necessary I can also work on Sundays too. During this time period, I intend to stay in touch with my mentor and ensure that I am going in the right direction.

- **Pre-GSOC and Community Bonding (Till 26th May):**
  - Work my way through Python/C API Reference Manual.
  - Review the existing built-in optimizations.
  - Research more on possible optimization for built-ins.

> *The following built-ins are in the form of*
> **`built-in(parameter(s)) -> returns : description`**

- **Week 1 (May 27-31):**
  - `all(num) -> (int, float, complex)` : returns absolute value of a number
  - `abs(iterable) -> (bool)` : returns true when all elements in iterable is true
- **Week 2 (June 3):**
  - `max(iterable, *iterables[, key, default]) -> (int, float)` : return largest element
  - `min((iterable, *iterables[, key, default]) -> (int, float)` : return smallest element
- **Week 3 (June 10):**
  - `map(function, iterable, ...) -> (map_object)` : Applies Function and Returns a List

  - **pow(x, y[,z]) -> (int, float)** : returns x to the power of y
  - **filter(function, iterable) -> (iterator)** : constructs iterator from elements which are true
- **Week 4 (June 17):**
  - **object() -> (object)** : returns memory view of an argument
  - **callable(object) -> (bool)** : checks if the object is callable
  - **divmod(x, y) -> (tuple)** : returns a tuple of quotient and remainder
- **Week 5 (June 24):**
  - **help(object)** : invokes the built-in help system
  - **memoryview(object)** : returns memory view of an argument
  - **zip(*iterables) -> (iterator)** : returns an iterator of tuples
- **Week 6 (July 1):**
  - **round(number[, ndigits]) -> (int)** : rounds a floating point number to ndigits places
  - **sorted(iterable[, key][, reversed]) -> (list)** : returns sorted list from a given iterable
  - **issubclass(object, classinfo) -> (bool)** : check if a object is subclass of a class
- **Week 7 (July 8):**
  - **raw_input()** : presents a prompt to the user
  - **input(string) -> (string)** : reads and returns a line of string

- **Week 8 (July 15):**
  - **`unichr()`** `: return the Unicode string of one character whose unicode code is the integer i`
  - **`reversed(seq) -> (iterator)`** `: returns reversed iterator of a sequence`
  - **`basestring()`** `: abstract type for superclass for str and unicode`
- **Week 9 (July 22):**
  - **`breakpoint()`** `: drops user into debugger at the call site`
  - **`delattr(object, name) -> (None)`** `: deletes attribute from the object`
  - **`reduce()`** `: apply function of two arguments cumulatively to the items of iterable`
- **Week 10 (July 29):**
  - **`property(fget=None, fset=None, fdel=None, doc=None)`** `: returns a property attribute`
  - **`cmp()`**`: compare the two objects x and y`
- **Week 11 (August 5):**
  - **`enumerate(iterable, start=0) -> (enum_object)`** `: returns a enumerate object`
- **Week 12 (August 12):** It will a buffer period in which I will try to clear my backlogs and add the API.
- **Final week (August 19):** This week I will be submitting my project.

4. Other Commitments

- I have my semester exams from 20th April to 10th of May.
- I might be unavailable on 28th May and 29th May.

5. Conclusion

I must mention that it has been a great learning experience contributing to **Nuitka** and the community was really helpful in getting me started with the development tasks. This project is really exciting to me because it will not only give me exposure to **Nuitka** codebase but also will increase my understanding of the inner workings of Python and specifically CPython implementation itself. People usually prefer C++ or Java for competitive programming but not Python because of performance issue. I believe once all the built-ins are optimized people will be more interested to use Python compiled with Nuitka for competitive programming. I am very enthusiastic to work on **Nuitka** with **Python Software Foundation** in the **Google Summer of Code 2019** and make some major contributions to the community.